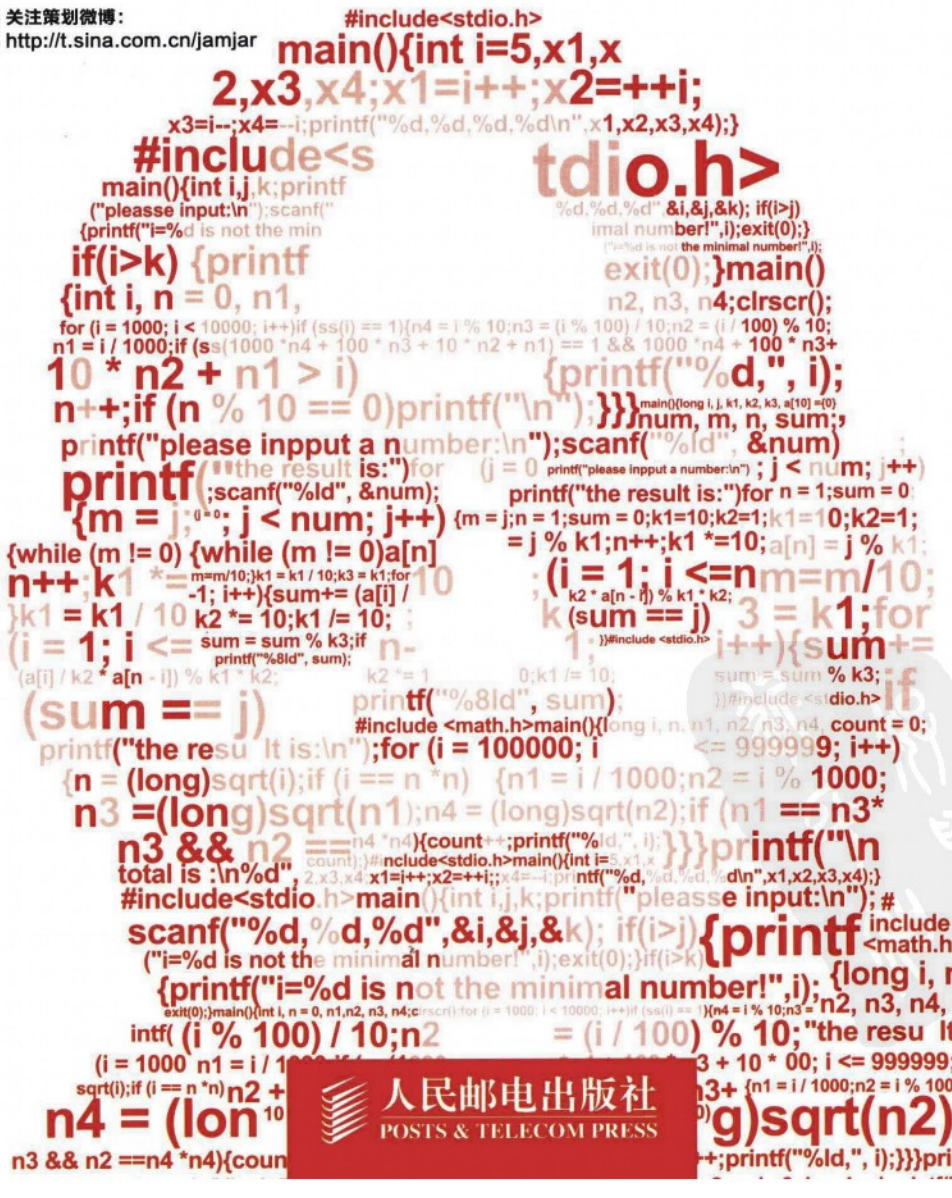
每多学一点知识
就少写一行代码

C语言编程之道

明日科技 刘彬彬 孙秀梅 李鑫 编著

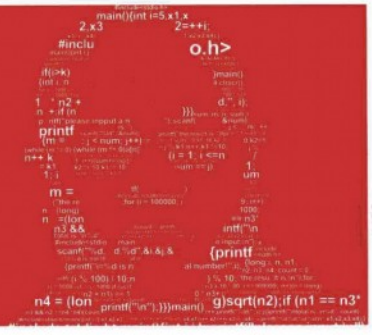
关注策划微博：
http://t.sina.com.cn/jamjar



人民邮电出版社
POSTS & TELECOM PRESS

每多学一点知识，就少写一行代码。

C语言编程之道



爱数组，爱结构体。

爱琢磨算法，爱玩弄内存，更爱掰着手指数字节。

爱自己冥思苦想，也爱和众生分享。

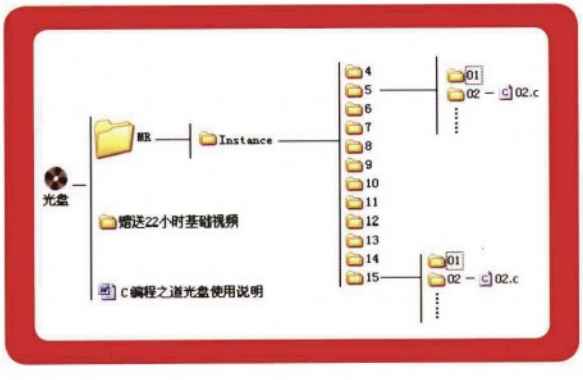
我不是无所不能的百科全书，也不是10块钱一本的如来神掌，

我是C语言编程之道。

别拿我和其他书相比，

我和它们不一样。

● **附赠光盘内容**



● **提升学习效率百分百**



ISBN 978-7-115-24546-5

分类建议：计算机／程序设计／C语言

人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

ISBN 978-7-115-24546-5

定价：59.00 元（附光盘）

C语言编程之道

明日科技 刘彬彬 孙秀梅 李鑫 编著

关注策划微博：
http://t.sina.com.cn/jamjar



人民邮电出版社
北京

图书在版编目（CIP）数据

C语言编程之道 / 刘彬彬，孙秀梅，李鑫编著. --
北京 : 人民邮电出版社，2011.3
ISBN 978-7-115-24546-5

Ⅰ. ①C… Ⅱ. ①刘… ②孙… ③李… Ⅲ. ①
C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第239923号

内 容 提 要

这是一本集编程感悟、技巧、陷阱、谬误和算法为一体的书籍，旨在引导读者如何正确地学习编程，帮助读者丰富编程经验。全书分为5篇，共15章，全面介绍了学习和应用C语言进行程序开发的各种陷阱与谬误、技术细节与经验技巧、常用的算法等。其涉及的技术主要包括C语言关键技术、指针、数组、数据结构、数学应用、算法等相关技术细节、技巧。

本书附带有配套DVD光盘，光盘中提供了书中全部实例和项目的源代码，这些源代码都是经过作者精心调试通过的，保证能够在Windows XP、Windows Server 2003操作系统下编译和运行。

本书非常适合大中专院校在校生、毕业生、求职者、编程爱好者学习，同时也可以作为想要学习编程的初学者的指导用书。

C语言编程之道

◆ 编　　著　明日科技　刘彬彬　孙秀梅　李　鑫
责任编辑　蒋　佳

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
大厂聚鑫印刷有限责任公司印刷

◆ 开本：800×1000　1/16
印张：24.25
字数：591千字　　2011年3月第1版
印数：1－3 500册　　2011年3月河北第1次印刷

ISBN 978-7-115-24546-5

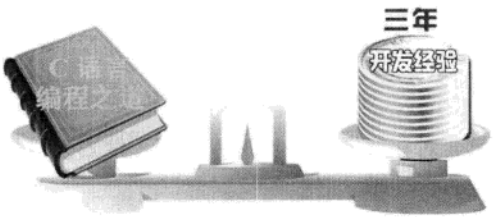
定价：59.00元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前　言

许多求职人员在面试时经常遇到的一个问题是“您有几年开发经验？”这对于一个刚刚走出校园的毕业生来说简直就是一个“原子弹”。“经验”真的很重要吗？是的，它是衡量个人能力的重要准则，它将许多求职者挡在门外。因此，经验真的很重要！

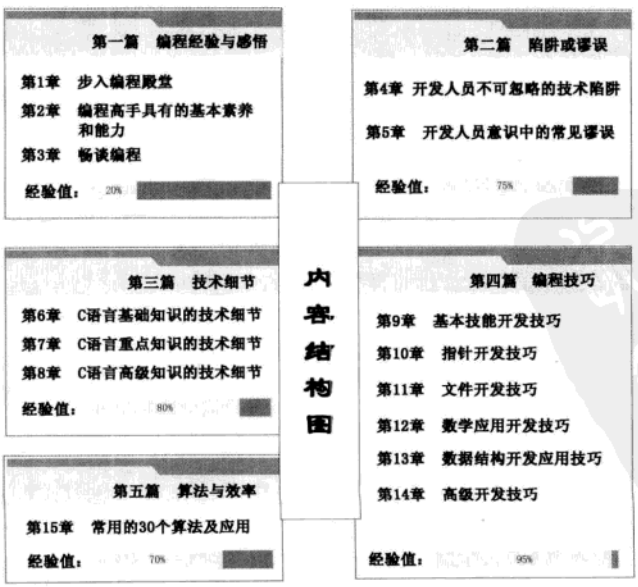
本书不是基础类书籍，不是项目、案例类书籍，而是一部丰富编程经验的书籍。它涵盖了 63 个编程细节、117 个编程技巧、22 个编程陷阱、36 个编程谬误、30 个优化算法。这些内容是作者结合自己多年的实际开发经验，精心筛选和提炼而成的。如果您想提高自己的编程经验，本书将是最佳的选择！



本书内容

本书分 5 篇，共 15 章。主要包括编程经验与感悟篇、陷阱与谬误篇、技术细节篇、编程技巧篇、算法和效率篇等内容。

本书内容结构如下图所示。



本书读者人群

本书非常适合以下人员阅读。

- C 语言求职人员
- 大中专院校的在校生和毕业生
- 相关培训机构的学员
- 语言已经入门，急需进一步提高的编程人员
- 编程爱好者
- 有多年开发经验的程序员

学习答疑

如果您在学习或使用本书的过程中遇到问题或疑惑，可以通过如下方式与我们联系。

- 服务网站：www.mingribook.com
- 服务 QQ：100310286（如要加入该 QQ，请标明是本书读者）
- 服务信箱：mingrisoft@mingrisoft.com
- 服务电话：0431-84978981/84978982
- 学习社区：bbs.mrbccd.com

我们承诺将在 5 个工作日内给您提供解答。

本书作者

本书由明日科技组织编写，参加编写的有刘彬彬、孙秀梅、李鑫、刘锐宁、梁水、李伟明、潘凯华、杨明、刘欣、李慧、董大永、李继业、尹强、张磊、赛奎春、高春艳、宋坤、安剑、巩建华、房大伟、吕双、刘云峰、王国辉、李钟尉、张振坤、陈丹丹、卢瀚、白伟明、王小科、梁晓岚、杨丽、顾丽丽、刘龄龄、王乐乐、杨明、朱晓、陈英、刘莹、李贺、肖鑫、王雷等。由于作者水平有限，疏漏和不足之处在所难免，请广大读者朋友批评指正。

明日科技

2010 年 12 月

目　录

第1篇　编程经验与感悟——站在前人臂膀上前行

第 1 章　步入编程殿堂……2
感悟 01　编程我也行……2
感悟 02　如何选择编程语言 ……2
感悟 03　培养对编程的兴趣 ……4
感悟 04　学习编程要打好基本功……6
感悟 05　多与别人交流 ……7
感悟 06　锻炼开放性思维……7
感悟 07　思维方式要灵活……9
感悟 08　不断温习学过的知识……9
感悟 09　坚持比激情更重要 ……9
第 2 章　编程高手具有的基本素质和能力 ……11
经验 01　善于总结编程经验 ……11
经验 02　创造或寻找方法……11
经验 03　重视程序中的小错误……12
经验 04　在调试与改错中成长……12
经验 05　养成良好的编程风格……13
经验 06　要重视实践练习 …… 14
经验 07　善于利用丰富的网络资源 …… 15
经验 08　学习与培训的必要性 …… 15
经验 09　多向他人学习…… 15
经验 10　写程序之前要分析…… 16
经验 11　写程序之后要总结…… 17
经验 12　软件需求规格说明书 …… 18
经验 13　逐步培养个人的编程思想 …… 18
经验 14　面试经验谈 …… 19
第 3 章　畅谈编程…… 21
感悟 01　程序员的成长之路…… 21
感悟 02　程序员的生存法则…… 22
感悟 03　程序员的生涯规划…… 23
感悟 04　项目开发经验谈 …… 24
感悟 05　信息技术改变我们的生活 …… 26

第2篇　陷阱与谬误——汲取前人的教训

第 4 章　开发人员不可忽略的 22 个技术陷阱 ……30
陷阱 01　忽略了大小写的区别……30
陷阱 02　“{}”与“()”使用不当造成错误……31
陷阱 03　多加分号导致出错 ……32
陷阱 04　使用循环语句不慎产生死循环……33
陷阱 05　忽视 do...while 循环次数产生的错误……34
陷阱 06　数据溢出产生的陷阱……35
陷阱 07　整数的除法导致结果的偏差……36
陷阱 08　变量未定义 …… 37
陷阱 09　输入数据时忘记添加&符号 … 38
陷阱 10　使用 scanf 函数输入格式不符 …… 39
陷阱 11　输入的数据类型不符的陷阱… 41
陷阱 12　switch 语句中忘记使用 break…… 42
陷阱 13　指针没被初始化 …… 43
陷阱 14　操作超出动态分配的内存 …… 44
陷阱 15　使用内存后未释放…… 45
陷阱 16　内存释放了继续使用 …… 46
陷阱 17　没有说明函数的头文件…… 47
陷阱 18　忽略了字符串结束符 …… 48

陷阱 19　数组长度使用变量 ……49
陷阱 20　数组下标“越界” ……51
陷阱 21　子函数未定义产生的陷阱 ……52
陷阱 22　子函数在说明和定义时
类型不一致……53
第 5 章　36 个开发人员意识中的常见谬误……55
谬误 01　Main 大写错误 ……55
谬误 02　“=”与“==”……55
谬误 03　scanf()错误……56
谬误 04　循环错误 ……57
谬误 05　在定义数组时，将定义的
“元素个数”误认为是可
使用的最大下标值 ……57
谬误 06　混淆了字符与字符串
常量……57
谬误 07　使用==比较字符串 ……58
谬误 08　没有使用零字符结束
字符串 ……58
谬误 09　不正确的使用 fgetc()等
函数……58
谬误 10　a 和&a 的区别……59
谬误 11　gets()函数的使用 ……60
谬误 12　忽略了变量的类型，进行了
不合法的运算 ……60
谬误 13　忘记加分号……60
谬误 14　路径名错误 …… 61
谬误 15　#include <stdio.h>和
#include "stdio.h"的区别…… 61
谬误 16　函数调用的错误 …… 61
谬误 17　空指针并非空字符串 …… 62
谬误 18　枚举类型容易出错的
几个问题 …… 62
谬误 19　&、|、~与&&、||、！的
谬误 …… 63
谬误 20　system()和 clrscr()的区别…… 63
谬误 21　使用 getchar()函数的误区…… 63
谬误 22　使用 errno 检测错误…… 64
谬误 23　宏定义中不可忽视的空格 …… 65
谬误 24　*p++是自增 p 还是 p 做
指向的变量…… 65
谬误 25　char a[]和 char *a 是一样的… 65
谬误 26　指针变量与零值的比较…… 66
谬误 27　const 和#define 是一样的…… 66
谬误 28　浮点变量与零值的比较…… 66
谬误 29　宏定义中的常见谬误 …… 67
谬误 30　a[i]=i++;代码正确吗 …… 68
谬误 31　数组和指针等价吗…… 68
谬误 32　指针和数组的区别…… 68
谬误 33　指针比数组执行快…… 69
谬误 34　++i 比 i=i+1 执行快…… 69

第 3 篇　技术细节——修炼编程内功心法

第 6 章　26 个 C 语言基础知识技术细节……72
细节 01　标识符的使用 ……72
细节 02　限定词 const……73
细节 03　限定词 volatile ……73
细节 04　变量的存储……74
细节 05　字符数据类型…… 74
细节 06　说明变量和定义变量 …… 75
细节 07　typedef 应用的技术要点…… 76
细节 08　使用 static 关键字声明
静态变量 …… 77

细节 09 如何区分“，”是运算符还是分隔符 ……78
细节 10 基本的算术运算 ……79
细节 11 自增、自减操作符的使用 ……81
细节 12 C 语言中的特殊表达式 ……82
细节 13 特殊表达式 e1?e2:e3 ……83
细节 14 使用 printf 函数输出% ……83
细节 15 %lf 与%f 差异 ……84
细节 16 scanf 与 gets ……85
细节 17 什么时候用 switch ……86
细节 18 if 语句的多个判断问题 ……88
细节 19 break 在 switch 语句中的使用 ……89
细节 20 while 与 do…while 的异同 ……90
细节 21 while、do…while、for 循环相互转换 ……91
细节 22 循环不能交叉 ……92
细节 23 goto 语句使用规则 ……93
细节 24 合理使用 return 语句 ……93
细节 25 合理使用空格使代码更规范 ……94
细节 26 C 语言如何解释 a < b < c ……94
第 7 章 19 个 C 语言重点知识技术细节……96
细节 01 判断数组中元素的个数 ……96
细节 02 向函数传递二维数组 ……97
细节 03 字符数组的结束标志 ……97
细节 04 extern 声明数组 ……98
细节 05 数组下标从 0 开始 ……101
细节 06 数组名作函数参数与数组元素做函数参数的不同之处 ……101
细节 07 ‘\0’ 和 “\0” 的区别 …… 102
细节 08 静态全局变量与全局变量 …… 102
细节 09 访问寄存器变量地址 …… 102
细节 10 参数校验 …… 103
细节 11 什么是空指针 …… 103
细节 12 if（p）检查指针是否为空 …… 104
细节 13 指针变量赋值相关 …… 104
细节 14 const 定义指针变量 …… 105
细节 15 char a[]与 char *a …… 106
细节 16 指针应用 …… 107
细节 17 指针类型和指针所指向的类型 …… 108
细节 18 指针运算与取地址运算符的联系 …… 109
细节 19 函数型指针 …… 109
第 8 章 18 个 C 语言高级技术细节……111
细节 01 定义结构体对结构体成员的类型限制 ……111
细节 02 成员运算符（.）和（->）的差异 ……111
细节 03 结构体和联合体的区别和联系 ……111
细节 04 enum 的好处 ……112
细节 05 .h 文件 ……113
细节 06 宏定义问题 ……114
细节 07 条件编译 ……115
细节 08 文件打开 ……115
细节 09 C 语言中的预处理功能 ……117
细节 10 使用宏时需要注意些什么 ……117
细节 11 文件包含的嵌套 ……117

细节 12 malloc()与 calloc()的区别 …… 118
细节 13 什么是“内存泄露” …… 119
细节 14 内存耗尽了怎么办 …… 119
细节 15 常见的内存管理错误有哪些 …… 120
细节 16 标准文件和一般文件有什么区别 …… 120
细节 17 C 语言的常用库文件包含哪些 …… 120
细节 18 为什么文件打开以后要及时关闭 …… 122

第 4 篇 编程技巧——程序员的神兵利器

第 9 章 C 语言开发人员必知必会的基本技能开发技巧 …… 124
技巧 01 十进制数转换为十六进制 …… 124
技巧 02 N 进制转换为十进制 …… 125
技巧 03 IP 地址形式输出 …… 126
技巧 04 数字由小到大排序 …… 127
技巧 05 阶梯问题 …… 129
技巧 06 判断闰年 …… 130
技巧 07 对调数问题 …… 131
技巧 08 评定成绩等级 …… 133
技巧 09 打印乘法口诀表 …… 134
技巧 10 打印杨辉三角 …… 135
技巧 11 特殊等式 …… 137
技巧 12 统计各种字符个数 …… 138
技巧 13 猴子吃桃 …… 139
技巧 14 删除字符串中连续字符 …… 140
技巧 15 求学生的平均身高 …… 141
技巧 16 灯塔数量 …… 143
技巧 17 老师分糖果 …… 144
技巧 18 小球下落问题 …… 145
技巧 19 彩球问题 …… 146
第 10 章 指针开发技巧 …… 148
技巧 01 使用指针实现整数排序 …… 148
技巧 02 使用指针实现数据交换 …… 149
技巧 03 指向结构体变量的指针 …… 151
技巧 04 使用指针输出数组元素 …… 152
技巧 05 用指针实现逆序存放数组元素值 …… 153
技巧 06 使用指针查找数列中最大值/最小值 …… 154
技巧 07 使用指针实现字符串复制 …… 156
技巧 08 使用指针实现字符串的连接 …… 158
技巧 09 使用指针实现字符串插入 …… 159
技巧 10 使用指针实现字符串的匹配 …… 160
技巧 11 用指向函数的指针比较大小 …… 162
技巧 12 使用返回指针的函数查找最大值 …… 163
技巧 13 用指针函数实现求学生成绩 …… 164
技巧 14 寻找指定元素的指针 …… 165
技巧 15 寻找相同元素的指针 …… 167
技巧 16 使用指针的指针输出字符串 …… 168
技巧 17 实现输入月份输出该月份英文名 …… 170

技巧 18 使用指向指针的指针对字符串排序 ……171
第 11 章 文件开发技巧 ……173
技巧 01 读取磁盘文件 ……173
技巧 02 将数据写入磁盘文件 ……175
技巧 03 格式化读写文件 ……176
技巧 04 成块读写操作 ……177
技巧 05 随机读写文件 ……179
技巧 06 复制文件内容到另一文件 ……181
技巧 07 合并两个文件信息 ……182
技巧 08 文件分割 ……183
技巧 09 文件加密 ……185
技巧 10 统计文件内容 ……186
技巧 11 创建文件 ……188
技巧 12 创建临时文件 ……189
技巧 13 重命名文件 ……190
技巧 14 删除文件 ……191
技巧 15 删除文件中的记录 ……192
技巧 16 查找文件 ……193
技巧 17 显示目录内同类型文件 ……194
技巧 18 关闭打开的所有文件 ……196
技巧 19 错误处理 ……198
第 12 章 数学应用开发技巧 ……201
技巧 01 求 100～200 之间的素数 ……201
技巧 02 可逆素数 ……202
技巧 03 回文素数 ……203
技巧 04 特殊的完全平方数 ……204
技巧 05 求 1000 以内的完数 ……205
技巧 06 三重回文数 ……206
技巧 07 亲密数 ……207
技巧 08 自守数 ……208
技巧 09 满足 abcd=(ab+cd)^2 的数 ……209
技巧 10 神奇的数字 6174 ……210
技巧 11 一数三平方 ……211
技巧 12 求等差数列 ……213
技巧 13 求整数的绝对值 ……214
技巧 14 正弦、余弦、正切值 ……215
技巧 15 自然对数的底 e 的计算 ……216
技巧 16 最大公约数及最小公倍数 ……216
技巧 17 求解二元一次不定方程 ……217
技巧 18 二分法求解方程 ……219
技巧 19 牛顿迭代法解方程的根 ……220
技巧 20 打印特殊方阵 ……221
技巧 21 求 3×3 矩阵对角元素之和 ……222
技巧 22 矩阵的加法运算 ……223
技巧 23 矩阵的乘法运算 ……224
技巧 24 打印 n 阶螺旋方阵 ……225
技巧 25 求车运行速度 ……226
技巧 26 卖西瓜 ……228
技巧 27 巧分苹果 ……228
第 13 章 C 语言开发人员必知必会的数据结构开发应用技巧 ……230
技巧 01 结构体简单应用 ……230
技巧 02 找最高分 ……233
技巧 03 平均成绩 ……235
技巧 04 比较计数 ……235
技巧 05 信息查询 ……237
技巧 06 计算开机时间 ……238
技巧 07 创建双向链表 ……240
技巧 08 创建循环链表 ……244
技巧 09 创建顺序表并插入元素 ……246
技巧 10 向链表中插入节点 ……248
技巧 11 从链表中删除节点 ……250

技巧 12　合并两个链表 ……253
技巧 13　应用栈实现进制转换 ……255
技巧 14　括号匹配检测 ……258
技巧 15　串的模式匹配 ……261
技巧 16　简单的文本编辑器 ……262
第 14 章　C 语言开发人员必知必会的高级开发技巧……267
技巧 01　设置 DOS 系统时间 ……267
技巧 02　设置 DOS 系统日期 ……268
技巧 03　函数实现字符匹配 ……269
技巧 04　查找位置信息 ……270
技巧 05　建立目录 ……272
技巧 06　复制当前目录 ……273
技巧 07　删除目录 ……274
技巧 08　访问系统 temp 中文件 ……276
技巧 09　设置组合键 ……277
技巧 10　获取当前日期与时间 ……278
技巧 11　设置系统日期 ……279
技巧 12　获取 BIOS 常规内存容量 ……281
技巧 13　获取 CMOS 密码 ……282
技巧 14　鼠标中断 ……284
技巧 15　备份/恢复硬盘分区表 ……286
技巧 16　获取系统配置信息 ……290
技巧 17　恢复内存文本 ……292
技巧 18　控制扬声器声音 ……295

第 5 篇　算法和效率——程序员的得力助手

第 15 章　常用的 30 个算法及应用 ……298
算法 01　冒泡排序 ……298
算法 02　选择排序 ……300
算法 03　直接插入排序 ……301
算法 04　归并排序 ……303
算法 05　希尔排序 ……305
算法 06　快速排序 ……307
算法 07　顺序查找 ……309
算法 08　二分查找 ……310
算法 09　分块查找 ……312
算法 10　哈希查找 ……314
算法 11　斐波那契数列 ……317
算法 12　角谷猜想 ……319
算法 13　歌德巴赫猜想 ……320
算法 14　四方定理 ……322
算法 15　尼科彻斯定理 ……323
算法 16　创建单向链表 ……325
算法 17　约瑟夫环 ……328
算法 18　双链表逆序输出 ……330
算法 19　二叉树的遍历 ……332
算法 20　用栈设置密码 ……334
算法 21　链队列 ……338
算法 22　SHA 加密 ……342
算法 23　MD5 加密 ……349
算法 24　RSA 加密 ……359
算法 25　DES 加密 ……361
算法 26　RC4 加密 ……366
算法 27　打渔晒网问题 ……369
算法 28　水池注水问题 ……371
算法 29　迪杰斯特拉算法 ……372
算法 30　百钱买百鸡 ……374

1

第 1 篇
编程经验与感悟
——站在前人臂膀上前行

第 1 章　步入编程殿堂
第 2 章　编程高手具有的基本素质和能力
第 3 章　畅谈编程



第1章　步入编程殿堂

历史的发展是无数先辈共同努力的结果，当代的我们可以说是“站在巨人的肩膀上”，用前人积累的成果，不断地创造着下一个辉煌。

软件开发的工作在今天常被视为一种体力活，我们拿着前人的成果可以更轻松地创造更有价值的东西。本人在软件开发行业摸爬数年，不敢说有多深厚的见解，但是也总结了一些软件开发学习的心得，希望与君共勉。

感悟 01　编程我也行

也许很多想要从事编程工作的人都会问，我真的可以胜任这份工作吗？其实这种想法的产生与计算机的发展历程有关。自计算机问世以来，软件开发就成为了令人羡慕的职业。在软件行业刚兴起的时候，程序开发被很多人追捧为高智商、高水平的高端技能，程序员就犹如高深莫测的武林高手般被世人崇拜敬仰，那时候的程序员充满了个人英雄主义色彩。计算机产业被誉为高新技术，计算机从业者被称为 IT 精英，让人觉得可望而不可及。

随着计算机的迅速普及，发展到今天，软件可以说无处不在，软件开发也不再是传说。而个人英雄主义时代也已经过去，当前的程序员，需要有高度的团队合作精神，一个庞大的系统是由众多程序员共同完成的。只要热爱编程并为之努力，编写出自己的程序并不是遥不可及的事情。你可以是一名程序员，你也可以只是出于兴趣，只要你喜欢，就可以进行编程。

有位编程前辈曾经说过：任何人都能编程，那么我们就开始自己的程序开发之旅吧。

感悟 02　如何选择编程语言

要想学习编程，第一件事就是要选择一个开发工具。“工欲善其事，必先利其器”，这是《论语》中的名句，大家在学习编程的时候都希望选择一门优秀的编程语言。现今国内外比较常用的编程语言不下几十种，选择一门适合自己的编程语言是学习编程的第一关，这涉及程序员的未来工作方向，甚至是职业生涯规划，笔者根据数十年的编程经验，并参考相关权威资料总结出以下几点。

- 选择与自己兴趣相近的方向

只有把工作当作一件快乐的事，才能使自己愿意为之奋斗终生，正因为这样，所以才有了中国

的医药学家李时珍在无病的情况下亲自尝试各种药材，才有美国著名的物理学家富兰克林在雷电天气里放风筝等名人轶事。兴趣是指你未来最希望从事的工作方向，根据自己的编程兴趣去选择编程语言，发挥自己最大的优点。接下来笔者列举了在软件开发行业中几个比较热门的设计方向，大家可以根据个人的兴趣来选择适合自己的编程语言。

- 嵌入式开发

这个方向的开发语言可以选择 C 或 C++等语言，它们灵活性好、兼容性强、效率高、能够适应比较底层技术的开发。

- 网络及 Web 应用开发

如今是互联网的天下，这个方向的开发语言比较多，比较流行的有 Java、C#、PHP、Python、Perl 等，其中 Java 是 JSP 开发技术的后台开发语言，C#是如今火热流行的 ASP.NET 开发技术的后台编程语言，Perl 的技术有些陈旧，有被 Python 取代之势。

- 应用软件开发

相对 Web 网络开发，桌面版的应用软件开发技术发展有些缓慢，但在我们的工作和生活中还经常用到，如 QQ、各种视频播放器等，这方面常见的开发语言或开发工具有 VC++、VB、Delphi、C#等。

- 数据库应用开发

数据库应用开发技术是许多编程语言必有的核心开发技术之一，常用的语言包括 Java、C#、VC++、VB、Delphi、PowerBuilder 等。

- 游戏软件开发

游戏开发目前多用 C、C++、Java 等编程语言。

- 尽量选择就业需求量较大的语言

选择编程语言时，应该查看一下近年来就业市场的需求情况，这个比较容易理解，例如，20 世纪 90 年代中后期，C/S 模式的数据库应用软件开发掀起浪潮，当时使用 PB、VB、Delphi 等编程语言开发的软件几乎占据了全部的市场，市场招聘异常火热，但时至今日，随着 Web 时代的来到，PB、VB、Delphi 等编程语言的程序开发人员招聘逐年萎缩。

- 尽量选择国际大公司所研发出的语言

选择编程语言及开发工具，建议选择国际大型软件公司所研发出的语言，因为其技术支持稳定、完善，而且它的战略发展直接引导着整个软件开发界的未来，所以使用它们的开发语言总体来说相对稳定，即使如 Sun 这样的大公司倒下了，但其开发语言 Java 仍经久不衰，更是后继有人（Sun 公司被 Oracle 公司收购），使用微软开发工具的程序员，更能够感受到微软公司长期给其带来的稳

定的技术保障，以及引领软件开发未来的编程思想。

- 选择现今流行且语法规范性强的语言

现今比较流行网络开发和面向对象的技术开发，那么选择如 C++、Java、C#等面向对象规范性相对较强的开发语言，是一个不错的选择，而 C#语言可以说是目前众多编程语言中面向对象最完善的一种。

C 语言是广泛使用的教学语言，具有超强的生命力。可以作为编程入门语言来进行学习。深厚的 C 语言知识基础可以为以后的程序学习和开发提供强大的思想准备。另外，我要说的是，不要过分讨论哪个语言好、那个语言不好，或者今天学这个，明天又学那个，这样终究只会一无所长。“十鸟在林，不如一鸟在手”，就算只用 C 语言，用好了，同样可以成为大师级人物。

感悟 03 培养对编程的兴趣

在学习编程和进行软件开发的过程中，总是要面对许多程序代码，因此对于初学者而言，学习编程可以说是一件非常枯燥的事情，为什么说对于初学者来说是枯燥的事情呢？这是由于初学者没有编程经验，只知道照着书本或资料去敲代码，面对诸多的程序代码必然会感觉乏味，而对于那些中高级程序员来说则不一样，他们不但有编程经验，而且编程的知识也已经系统化，能够编写出所要实现功能的程序代码，如编个小游戏，做一个实用的软件等，所以编程对于那些中高级程序员来说是一种乐趣。

说明： *之所以对初学者与中高级程序员进行对比，只是为了说明兴趣的重要性，并不是说初学者面对那些枯燥的代码就不能学好编程，因为很多程序员都是从这个过程走过来的。*

既然众多的代码令人厌烦，那么初学者应该想办法弥补由于程序代码所带来的烦恼，最好的办法就是在编程的过程中加入一些身边的或者曾经经历过的感兴趣的内容，比如，可以将原来输出的字符串，改为输出一个特殊的符号或图形，原来输出的是一行星号，可以将其改为输出一个由星号组成的三角形、棱形等，久而久之，对编程就会产生兴趣，而不会再觉得编程是一件繁琐的事情了。兴趣是最好的老师，所以一旦对编程产生了兴趣，学习编程也就会觉得轻松了，只要是自己感兴趣的，即使遇到困难也会以解决困难的过程而感到高兴。



图 1.1 扫雷游戏

培养编程兴趣的另一种好办法，就是多留意身边的一些软件，比如 Windows 提供的扫雷游戏，如图 1.1 所示，通过自制这样的游戏既可以开发智力，又可以提高对编程的兴趣。

再来看一个 Windows Vista 和 Windows 7 自带的 Purble Place 游戏，该游戏由 3 种游戏 Comfy Cakes、Purble Shop 和 Purble Pairs 组成。

其中 Comfy Cakes 游戏将你置于 Purble 厨师的面包店中，你的工作就是完成一批蛋糕的订单。由于面包店的客户非常特殊，因此需要采用正确的形状、口味、夹心、糖衣和点缀组合制作蛋糕，如图 1.2 所示。

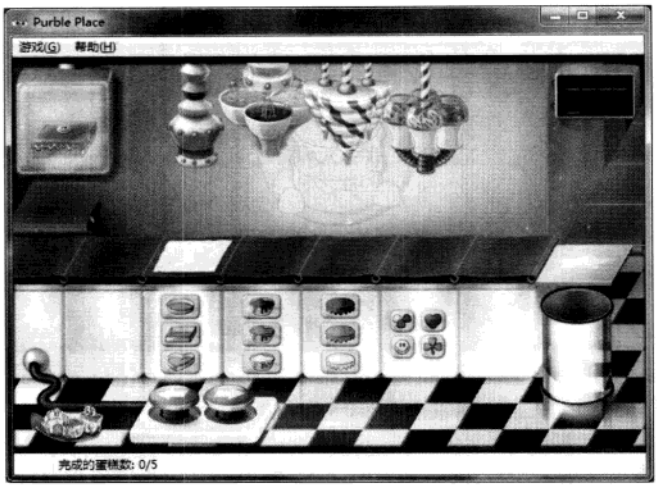


图 1.2　Comfy Cakes 游戏

Purble Shop 游戏用于测试推断能力。游戏目标就是使 Purble 的脸与窗帘后面的神秘 Purble 相匹配。从架子上选择头发、眼睛、鼻子和嘴唇，如图 1.3 所示。

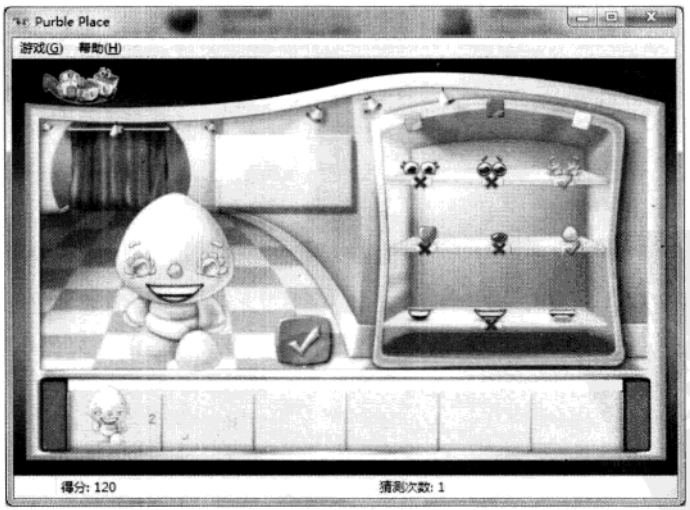


图 1.3　Purble Shop 游戏

Purble Pairs 游戏的目标是在时间用完之前找到所有完全相同的图片。通过翻转一个图块以显示图片，然后尝试找到与其配对的图片。使用“秘技”点可以对整局游戏进行浏览，如图 1.4 所示。

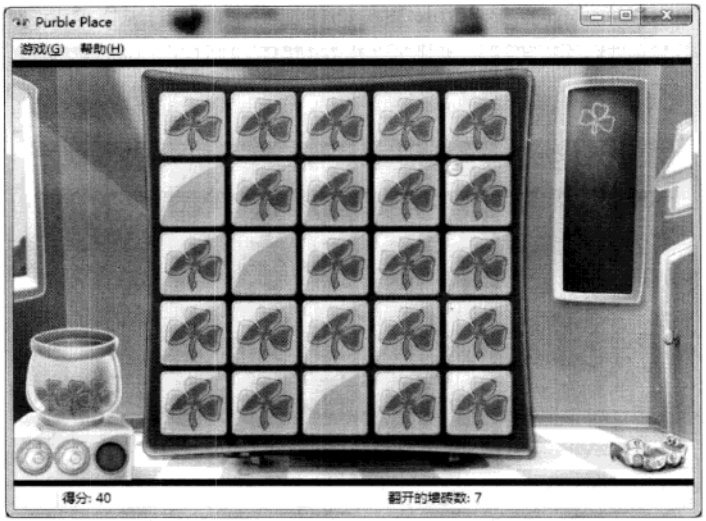


图 1.4 Purble Pairs 游戏

说明： 通过身边的这些小游戏和小软件，完全能够使你对编程产生兴趣，因为这些小游戏也好，或是一些实用的软件也罢，都是通过程序语言编写出来的，只要有了兴趣，自然也就能够学好编程了。

感悟 04 学习编程要打好基本功

万丈高楼平地起，空中楼阁不存在，打好基础是你勇攀高峰的基石。有些编程的疯狂爱好者简单看看语法之后，马上就投入到大型实例项目的学习中，然后仿照实例开发实际项目。在这些人中，确实有一部分人通过学习实例学会了编程，但这些人在编程过程中遇到的各种困难和障碍只有他们自己心里最清楚。要真正掌握好一门编程语言的精髓，学好其语言基础及其运行机制与原理是十分重要的，只有扎实地掌握了这门语言的基础知识，才能够比较快速地提升自己的编程技能。

可以说编程离不开基础知识，就拿条件语句 if 来说吧，可以说它在编程的过程中是无处不在的，也正是由于条件语句的存在，才能很好地控制程序的业务逻辑，否则就很难对程序进行控制，程序代码就会按顺序从上到下一直执行下去，直到所有代码都执行完，而 if 语句又有多种用法，这就要根据需要选择是使用简单的 if 语句、复杂的 if 语句还是嵌套的 if 语句。

注意： 基础知识对于每个程序员来说都是非常重要的，如果能熟练掌握并灵活应用基础知识，对于以后的工作和学习都是非都有帮助的，因此必须要学好基础知识。但是，并不是说刚开始学习编程时就能够熟练掌握那些基础知识，可以在学习一段时间后，再回过头来看一下那些基础知识，这样会更容易掌握，理解会更深刻。

实际上还有许多基础知识都是编程必须掌握的，如基本数据类型、常量、变量等，也是编程过程中必须掌握的，要使用常量和变量就要使用基本数据类型进行声明，掌握了使用基本数据类型声明常量和变量，对于使用类声明对象也就很容易掌握了，因此在编程中那些基础的知识往往都是很重要的。

感悟 05　多与别人交流

在学习编程的过程中，经常需要看一些书籍，或上网查一些资料，而有些知识可能比较难以理解，这时可以向别人请教，不要放不下面子，古人尚且知道“不耻下问”的道理，更何况我们是在学习，因此更应放下面子，向别人虚心请教，这样才能做到“知之为知之，不知为不知，是知也”。

可以说交流是提高技能最快的制胜法宝，一个人的思路往往是狭隘的、局限的，甚至是偏激的。这就要多交流，每个人的思维方式不同、角度各异、各有高招，通过交流可以不断的吸收别人的长处，丰富的编程经验，快速提高自己的编程水平。“三人行必有我师”，也许在一次和别人不经意的谈话中，就可以迸出灵感的火花。

交流的方法很多，形式各异。

总之，交流是能力迅速提高的法宝。所以在学习编程的过程中一定要多与别人交流，通过交流可以做到取长补短，相互进步，这样不但可以提高自己的技术水平，同时还可以提高对编程的兴趣，而且还可以增加一些志同道合的编程好友，何乐而不为呢？

在与别人交流时，要将自己的一些好的东西分享给别人，这样别人也会被你的真诚所打动，并乐于将好的技术分享给你，这样就可以学到更多的知识，对于自己今后的编程工作也是非常有帮助的。

说明： 对于编程时遇到的不会的内容，不要马上向别人请教，可以自己先多思考，能自己解决的尽量自己解决。如果问题实在解决不了，再向别人请教，并且在向别人请教时，一定要将问题说清楚，对于输出的一些异常信息，最好也要给出，这样可以方便别人对所出现的问题进行分析。

感悟 06　锻炼开放性思维

一个高级程序员应该具备开放性思维，从里到外的所有的知识都能了解。然后，看到世界最新技术就能马上掌握，马上了解。实际上，技术到达最高境界后，是没有分别的。任何东西都是相通的，只要你到达这个境界以后，什么问题一看就能明白，一看就能抓住最核心的问题，而不会被其他的枝叶或表象所迷惑，做到这一步之后才算比较成功。

从程序员本身来说，如果他到达这一步以后，他就已经形成了开阔的思维，他就能做战略决策，这对他将来做任何事情都有好处。

锻炼开放性思维首先要有一个宽广的知识面，涉猎到的计算机知识要非常广泛，最后还要精通程序的开发过程。否则只知道用开发语言去开发程序知识片面的、狭隘的。当程序中遇到稍微底层的内容就会束手无测了，而只有了解一系列的计算机知识体系结构，才能遇到各种问题都能够得心应手，坦然面对。

说明： 锻炼开放性思维，首先要有一个非常广阔的知识面领域，这样在考虑问题的时候才能更加全面。思路更加开阔，方法也就更加多元化，思维也就更加敏捷。

- 了解 Windows 平台的运转知识

任何一个软件一定都是和一个平台相关联的，游离于平台之上的系统和软件是不现实的，这样的软件几乎都是不能用的。比如当你对操作系统平台的知识非常了解的时候，在思考一个问题时，就能马上想到操作系统能提供哪些功能，自己在做些什么功能就能够将任务完成。

- 掌握多种语言，放宽思维

不同的程序语言有它适合的不同领域，当你面对某个问题时，需要找一些合适的语言来解决这些问题。花时间去探索其他你所不熟悉的语言，能让你的思路更加广阔，思考的问题也更加全面。

- 掌握拆分思想

举个例子：在编写程序时，并不是将所有的内容都放在主函数中。为了方便规划、组织、编写和调试，一般的做法是将一个程序划分成若干个程序模块，每一个程序模块都完成一部分功能。这样不同的程序模块可以由不同的人来完成，从而可以提高软件开发的效率。在编译时，如果发现错误，可以在模块的范围内进行改正。在程序运行过程中，主函数可以调用其他的函数，其他函数也可以相互调用。

这种拆分思想带来的好处是很大的，分层分的就是职责，一分层力度就细，力度一细复用性就高了，编程的时候尽量让类和函数的职责单一，这样内聚性就高了。类之间或函数之间的关系不密切，耦合度就低，这样会带来强大的扩展性。

上面对锻炼开放性思维举了 3 个例子，读者要尝试锻炼自己的开放性思维，不要拘泥于固定的思维方式，遇到问题的时候要多想几种解决问题的方案，试试别人从没想过的方法。丰富的想象力是建立在丰富的知识的基础上，除计算机以外，多涉猎其他的学科，如天文、物理、数学等。

说明： 比尔·盖茨曾说，他招人的时候宁愿招一个学物理的，而不是学编程的。学物理的人会有非常广的思维，他考虑的事物小到粒子、大到宇宙，思维空间非常广阔，因此他思考问题的时候就会更有深度。

感悟 07　思维方式要灵活

做事不能钻牛角尖，编程更是如此。编写程序不仅仅是在敲代码，更是在编写一种逻辑思维，让这种逻辑思维与现实的业务工作相吻合。在编写程序的过程中，要用到你所学到的编程知识和技能，编写出优秀并且能够解决实际问题的代码，是程序员不变的追求，可事非人愿，实际上程序员在编程过程中经常碰壁，这就要求程序员必须灵活运用所掌握的编程知识和技能来解决实际问题。

许多程序员在编写程序的过程中都有过这样的经历，遇到一个进退两难的技术问题，明明使用某种技术就可以解决，但其他的问题又出现了，用了好长时间也解决不了，起身走走或别人的一句话，马上茅塞顿开，发现自己钻了牛角尖，所以只要把思维方式变通一下，就可以很容易地解决某些问题，所以我们要在工作和学习中不断地培养灵活的思维方式和发散思维能力，以提高我们解决问题的能力。

思维方式灵活其实依赖于勤于思考、积极的思考。遇到问题要多角度思考，即使有现成的解决办法，也要动脑想一想有没有更好的其他办法。

感悟 08　不断温习学过的知识

孔子说：“温故而知新，可以为师矣。”这句话流传至今仍行之有效，也就说明了它的实践意义。广泛的阅读典籍，温习学过的知识，从中可以得到新的感悟，以达到融会贯通的效果，这样就可以称得上老师。所以要求编程者温故而知新也是有道理的，不但可以防止由于长时间不使用对知识的遗忘，还可以在“温故”的同时“知新”，岂不一举两得。

温故知新的道理不用我多说，相信大家都懂，就看你如何去做了。

感悟 09　坚持比激情更重要

坚持就是胜利，这是一句老话。这句话理解起来很容易，但是要想做到是非常不容易的。学过编程的人大多数都会有这种体会，在这里不妨把它揭露出来，在学习编程的初期，多数人都有“心潮澎湃”的那种激情，对编程充满了神秘和向往已久的那种感觉，更有人会暗下决心把编程作为自己一生要为之奋斗的事业来做，但谁料到编程是山高路远，看看身边曾经在一起学习过编程的人（包括在学校的同学），还有多少人在与我们并肩作战，应该是不多了吧！

计算机程序设计能给人以激情和自我实现的成就感，但编程是一种比较抽象和复杂的工作，需要掌握大量的基础知识和编程技巧，尤其是面向对象的编程，抽象性很强，有些人遇到了困难就打了退堂鼓，浪费了精力不说，更浪费了许多宝贵的学习时间，激情是一时迸发的，而坚持才是你永

久要面对的。

计算机软件开发技术更新非常快，毫不客气的说，两年不学习新知识，你原有的技术就已经快被淘汰了，所以对那些已经是程序员的人来说，坚持学习也非常重要，程序员要是能够保证每天都学习，那么个人技能和个人职业生涯都会得到不错的发展，至少不会被行业淘汰。



第2章 编程高手具有的基本素质和能力

在计算机编程领域里，每一种编程语言的内容都博大精深，而且大部分编程语言的技术都在不断地更新，尤其现今流行的面向对象的编程语言，其新知识新技术更新更加频繁，让程序员们应接不暇。仅仅学会编程基础知识或做一些简单小型项目，还不足以让你在编程世界里立足这就需要我们在工作和学习过程中不断地提高个人的编程技能。

经验01 善于总结编程经验

自我总结，是提升个人技能的重要手段，就像上台阶一样，不断总结经验和教训，不断提升自己的技能，然后编写出更加优秀的代码，当然因实际工作情况和个人素质的制约，技能的提升可能会遇到瓶颈，但经验总结总是有百利而无一害的，而且不受任何限制。

作为程序员可能都有过这样的感觉，项目忙没时间总结，既然开发技巧都写在项目里，需要时现看代码就行了。笔者当年也是如此想过，可时间一长，随着项目数量的增多，需要时真的很难找到。项目忙，那是说辞，时间就像有个名人说过的那样，是可以挤出来的，在将来用到经验技巧时，你就会深深体会到事半功倍这句话的含义了。

在进行编程经验总结时，不但要总结经验技巧，还要学会总结失败与教训。总结经验不但可以对旧知识有更深刻的理解，而且还会发散个人的思维，这样个人的技能会显著地提高。

经验02 创造或寻找方法

懒人造就方法，愚公移山的精神固然值得称赞，但是人的精力是有限的，如果使用巧妙的方法来实现一件事情，就会节省不少时间。也就是说，学习工作中，要勤于思考，使用巧妙的方法来解决问题（但不是投机取巧）。这使我想起了一位朋友，也是从事软件开发，技术还是不错的，他就是一个典型的“懒人”。例如，工作中有时候需要整理大量的文档，这位仁兄说做这些真麻烦费力，就做了一个文档处理的程序，几十篇的文档，在电脑上自动打开，提取信息，然后自动关闭，只需几分钟时间，而操作者只需按几个按钮，然后就可以喝着咖啡，看着电脑上的程序自动将文稿整理完成。这位朋友在编写程序的时候也是懒人，总是使用简单的代码或是想方设法找到简单的方法实现自己想要实现的功能。这也是编程者应该具有的素质，就像很多人经常说的——简单的就是最好的。学习编程的时候可以不用完全如此，但是勤于思考是必须要的。在思考中常常会得到解决问题

的更好的办法。

经验 03 重视程序中的小错误

程序出错是在所难免的，在 C 语言中经常会提示一些警告信息。这些警告并不影响程序的正常运行，但是大家千万不要不负责任的对其置之不理。这样会使程序留下隐患，并且可能后患无穷。当编译器提示警告或者错误信息的时候，要认真分析这些提示信息，并找到出错根源，以解决问题。但关键的是要记住这些问题，并在以后编程中多加注意，这也是编程经验的一种积累。而且这样往往会从错误中发现很多问题。对错误的研究会加深对知识深度的挖掘。

经验 04 在调试与改错中成长

调试可以说是每个程序员都必须掌握的技术，它伴随着程序开发的整个过程。学过、钻研过程序设计的人都会有同样的感受，在很多情况下，调试程序的过程会比程序编写的过程更加困难。任何一个程序员都不敢说，他编的程序是完全正确的。几乎每一个稍微复杂一点的程序都必须经过反复地调试、修改，最终才能完成。所以说，程序的调试是编程中的一项重要技术。

说明： 所谓程序调试，就是将编写的程序投入实际运行之前，用手工或编译程序等方法进行测试，修正语法错误和逻辑错误的过程。

程序调试可以说就是遇到问题、并且解决问题的过程。

- 程序设计中的常见错误

在程序设计中，容易出现的错误主要有 3 种类型：编译错误、运行时错误和逻辑错误。

（1）编译错误

编译错误是指程序在编译过程中出现的错误。它是由于不正确地编写代码而产生的如非法使用或丢失关键字、遗漏了某些必需的标点符号、函数调用缺少参数或传递了不匹配的参数等。

（2）运行时错误

运行时错误是指应用程序在运行期间执行了非法操作或某些操作失败，如打开的文件未找到、磁盘空间不足、网络连接断开、除法中除数为零等。

（3）逻辑错误

逻辑错误是指应用程序未按照预期的方式运行时所产生的错误。一般来讲，这不是属于语法层次的错误，应用程序可以执行，但是得不到正确的预期结果。

提示： Bug 是你的过错还是别人的过错，这一点并不重要，重要的是及时修正问题，而不是追究责任。

经验 05　养成良好的编程风格

作为一名高级程序员，所编写的程序应该具有良好的风格，也就是说程序应该规范、易于理解、具有较好的可读性，并且所编写的程序还要模块化，并为程序代码添加注释和完善的异常处理机制以及说明文档等。因此写程序并不是简单地写代码，也不是为实现程序的功能而写代码，如果简单地为实现程序的功能而写代码，这样的程序就不能说是一个好的程序。而真正优秀的程序，则应做到所编写的程序具有良好的风格，让别人容易理解和接受。

说明： 这里一再强调程序的风格，并不是说程序的功能不重要，这里只是说明作为一个优秀的软件，它的风格更重要而已。当然，话又说回来了，编程风格再好，如果所编写的程序并不能满足用户的需求，也就是说程序所实现的功能并不符合用户的要求，那这个程序一样是个失败的程序，编程风格和程序的功能二者都是缺一不可的，只是相比之下，编程风格更重要而已，程序并不是编写完就结束了，因为后期还要维护，这就要求所编写的程序要具有良好的风格，使别人能够理解和接受，并具有较好的可读性。

程序开发完成以后，还需要进行维护，因此，所编写的程序应具有较好的可读性，方便对代码的理解和维护，所以要形成良好的编程风格，主要应注重和考虑以下几点。

1. 要有较好的可读性

（1）使用有意义的标识符：标识符的命名应具有一定的意义，最好用英文单词，这样可以增强程序的可读性，方便代码的维护。

（2）要为程序添加注释：正确的注释能够帮助读者理解程序，尽量每条语句、每个方法和类等都要添加注释，并说明语句、方法和类等的作用，这样有利于对代码的理解和维护。

（3）代码结构要清晰：为使代码的结构清晰，可以在程序中利用空行、空格、制表符等使程序结构层次分明，也就是说要对程序代码进行分段和添加缩进。

2. 语句的结构应简单易懂

语句的结构应该简单直接，不应该为提高效率而把语句复杂化。因此需要注意如下几点。

（1）一行只写一条语句，这样的程序结构清晰、易于理解。

（2）除非对效率有特殊要求，否则程序编写应优先考虑的是清晰性，即增强可读性。可以通过添加缩进、分段和注释等，以使代码可以被别人理解。

（3）避免不必要的转移，大量的代码转移，会增加程序的复杂性，不利于对代码的理解。因此，如果代码不是很多而且不是很复杂的情况下，应尽量做到一个功能只在一个方法中实现。

（4）避免采用复杂的条件语句和带有否定条件的条件语句，这样会降低程序的可读性，不利于对代码的理解和维护。

（5）程序要模块化，在编写程序时尽量做到模块功能单一化，也就是一个模块只实现一个功能，并利用信息隐蔽技术，确保每一个模块的独立性。

说明： *在整个软件生命周期中，所花费用最多的就是软件维护阶段，因此，对于一个优秀的程序，要求代码具有较好的可维护性和扩展性，就要求程序代码的结构应简单清晰、易于理解，这样可以方便程序地维护和扩展，从而减少维护费用和维护时间。*

3. 输入和输出应方便用户操作

输入和输出信息是用户直接接触的，可以说系统能否被用户接受，在很大程度上往往取决于输入和输出信息的风格。因此在进行程序开发时，输入和输出应尽可能方便用户的使用，所以在设计程序时应考虑如下原则。

（1）对所有的输入数据都要检验其合法性，并对不合法的输入给出提示，并说明为什么不合法，应该输入什么样的数据才是合法的等。

（2）输入格式要简单，这样可以使输入的步骤和操作尽可能简单，从而可以大大方便用户的使用。

（3）通过使用缺省值，可以方便用户输入数据，提高用户的工作效率，因此对于可以设置缺省值的字段，应尽量设置缺省值。

（4）在进行数据的批量操作时，要保障数据的完整性和一致性，比如对数据库进行批量操作时，可以使用事务来保证数据的批量操作，要么都成功，要么都不成功。

（5）在以交互方式输入数据时，应使用标签等说明符明确提示用户应输入的信息，并在删除、修改和保存数据时，使用对话框在屏幕上给出提示信息和确认信息，以使用户能够了解所进行的操作是否成功。

经验 06 要重视实践练习

实践出真知，相信这句话大家都不陌生，学习编程也一样，不但要眼勤，也要手勤，通过实际动手编写代码可以使你更加深刻地理解技术要点。而且即使理解仍不够深刻，也能增强你对知识点的印象。编写代码后运行程序并不断地调试程序，这样你会对知识点有更多和更深层的理解。

有些程序员特喜欢看书，而且看得很快，但懒得动手，看完一本书后开始做项目，发现自己写代码非常困难，简直到了寸步难行的境地，这其实就是缺少实践练习的原因。当你仅仅把大量的知识灌入大脑中，而没有通过实践练习把这些知识点有机地结合，将会很难写出符合项目要求

的代码。

实践的重要性在当今程序员的就业方面体现得非常明显，我们不妨到各大招聘网站上看一看，招聘程序员几乎都要求有工作经验，更有甚者要求 3～5 年以上的工作经验，针对这种情况，某些 IT 技术培训学校更是大力推出“仿真项目实战练习”的课程，这些种种现象正是说明了实践练习的重要性。所以有些业界的同仁通过长期的开发工作总结出这样的道理：做项目是学习编程技能的最好方法。对于这句话，不能说它完全正确，但它确实体现了实践的重要性。

经验 07　善于利用丰富的网络资源

现如今是网络时代，网络信息非常丰富，在生活中遇到问题或困难，都可以通过搜索引擎来查找相关的信息。网络或多或少给我们一些提示，而编程资料就更加丰富了，专业的编程技术网站、编程技术论坛、程序员的博客等，都可以给我们提供丰富的学习资料，而且这些资源都是先行者的经验和教训的总结，尤其是某些高级程序员的博客文章，其含金量非常之高，因为他们总结的往往是真实的在大型项目中遇到问题的解决方案，是我们在书上甚至短期工作中无法得到的宝贵经验，我们可以收藏起来或转到自己的博客之中，然后细细品味。在看别人博客的同时也要善于自己写博客，把自己的经验技巧写入博客中，既可以总结经验，也可以与网友们分享技能，同时网友也会发表自己的看法，相互交流，从中不断提高自己的编程技能。

经验 08　学习与培训的必要性

俗话说，学无止境。活到老学到老，从事各个行业的工作都需要不断地学习，才能不断地提高自己的工作能力。软件开发是现今各种行业中知识更新速度最快的行业之一，真可谓不进则退，学习不但能够减少你在编程领域的技术盲区，更是掌握新知识的最好手段，程序员若不学习，可能很快就要被行业淘汰，所以学习对于程序员来说是至关重要的。

向编程技能的金子塔顶端攀登，几乎是所有程序员的梦想，每向“金字塔”的顶端迈进一步，都需要更加丰富的编程知识和更加高深的编程技能，这除了依靠个人的自身素质外，更需要程序员坚持不懈地学习以打好每个阶层的基础，从而不断地向顶端攀登。

学习的方式有多种，可以在工作之余自学，也可以参加高级培训班给自己充电，更可以在项目开发过程中不断积累经验。

经验 09　多向他人学习

三人行必有我师，这是论语中的名句，说的就是要善于向别人学习，学习编程也一样，不仅可以向别人学习知识，而且也可以向别人学习解决问题的方法，与他人交流可以丰富你的知识、拓展你的思维、加深你对知识的理解并增强团队的合作意识。

在实际工作中，我们经常会碰到这样一些程序员，他们少言寡语，极少与人交流技术，善于自

己琢磨，甚至自言自语，那么这些人在进行团队开发时就会出现问题，遇到疑难问题时解决缓慢，即使与其沟通也并不顺畅，固执己见，团队而合作意识差，进而影响整体项目的开发进程。

下面通过一个图示来形象说明善于与他人进行交流沟通对提高编程技能的重要性，如图 2.1 所示。

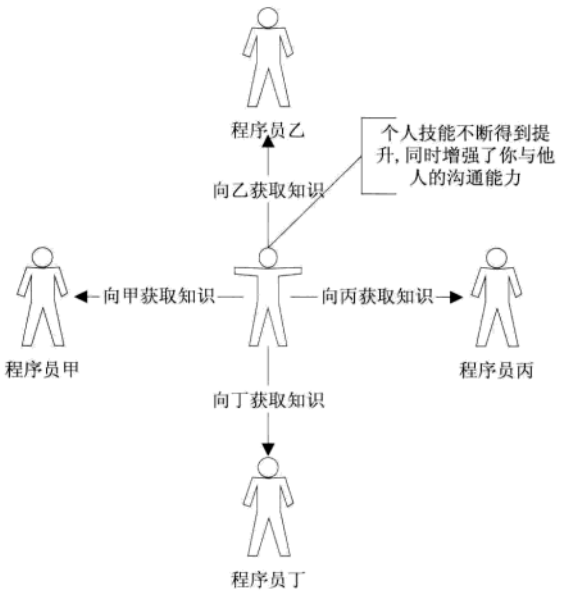


图 2.1　编程技能交流图

说明： 当然万事都不可走极端，也不可以遇到问题就问别人，自己不动脑，要将善于自我思考与相互交流结合起来才行。

经验 10　写程序之前要分析

在编写程序之前，项目负责人和需求分析人员应该适当了解行业背景，如果你是这方面的行业专家，那就最好不过了，这非常有利于对系统的分析，通过对用户的需求进行分析，确定目标系统所要实现的总的功能，然后再对系统的功能进行细分，将其分解为若干个子功能，并画出系统的功能结构图。

有了系统的功能结构图，接下来需要让用户对各功能进行确认，如果没有通过用户的确认，说明分析的结果不能满足用户的需求，因此，还要继续对用户的需求进行分析，并与用户共同完成系统功能结构图的设计。如果功能没有通过用户的确认，以后其他的工作将无法进行。

说明：在开发程序前一定要与用户进行沟通，并根据用户的需求对整个系统的功能进行分析，对于最终分析的结果，也就是整个系统的功能，一定要再让用户确认一下，这样可以防止功能遗漏或者出现部分功能不能满足用户需求的情况，同时也可以避免为以后的开发工作带来麻烦，因此，与用户的沟通很重要。

系统的功能确定之后，就可以根据用户的需求和系统的功能设计数据库了，在设计数据库时，要根据系统的功能和用户的硬件环境选择合适的数据库管理系统，并根据系统的功能设计出数据库中表实体 E-R 图，并设计出功能完善的数据库表和视图等数据库对象。

注意：数据库设计是一个非常重要的过程，数据库设计的好坏将直接影响整个系统的开发过程，如果数据库设计不合理或出现问题，将会影响整个系统的开发进度，甚至返工。因此，在需求分析之后，必须要将数据库设计作为重中之重，这样才能快速高效地完成整个系统的开发。

系统功能和数据库设计完之后，并对项目的相关需求都进行了分析，项目负责人就可以将任务分配给程序开发人员进行项目开发，在开发的过程中也要尽量与项目负责人和用户进行沟通，做到及时发现问题，及时解决问题，开发出真正优秀的产品。

经验 11　写程序之后要总结

程序开发完以后，还要多思考，进行归纳和总结，将程序的开发过程和最终软件产品的优点和缺点进行记录，对于优点，要在以后的开发过程中多借鉴，对于其中的缺点，要进行完善，做到精益求精，使所开发的软件能够真正满足用户的需要。

每完成一个软件产品的开发，都要进行归纳总结，随着开发软件产品数量的增加，所归纳总结的内容也就越多，这些内容达到一定量，那就是别人想花钱都买不到的宝贵财富（即工作经验），有了这些知识的积累，对于以后的工作和学习都是非常有帮助的，不但容易找工作，而且还会得到公司领导的重用。

说明：写程序之后进行归纳总结，是每个程序员都应该做的事情，说到底无非就是做个笔记，将一些好的、优秀的内容记录下来，并对那些不好的内容进行更正，并记录更正前与更正后的区别，只要养成这样一个好的习惯，不远的将来就一定会成为一名经验丰富的编程高手。

总之，在完成软件产品的开发之后，一定要进行归纳总结，对于其中的优点，要继续发扬，对于其中的缺点，要从中吸取教训，做到以后不再犯同样的错误，最终将自己培养成为一个具有丰富经验和远大报复的高级程序员和软件工程师。

经验 12 软件需求规格说明书

软件需求规格说明书是需求分析的产物，是软件开发中的重要参考文档，可以方便系统分析员、程序开发人员以及用户之间的交流与合作，可以确定目标系统的功能和其他一些与用户需求有关的信息，是进行软件开发、测试和验收的依据。

软件需求规格说明书大体包括以下内容。

1. 概述

概述部分用于对软件系统的目标和任务进行描述。

2. 数据描述

数据描述部分是对软件系统所要解决的问题进行详细说明，从而可以加深开发人员对所要解决问题的理解。

3. 功能描述

功能描述部分是针对软件所要实现的功能，应该给出的处理方法以及需要考虑的限制条件，是进行软件开发的依据。

4. 性能描述

性能描述部分用于对软件的性能、应满足的条件、测试方法和需要考虑的特殊问题等进行描述，是进行软件开发、验收和确认的依据。

5. 附录

附录中给出了一些补充资料，如词汇表、分析模型和参考文献等，其中词汇表中可以包括词头、缩写等内容；分析模型可以包括数据流图、类图及实体—关系图等信息；参考文献包括技术参考资料、产品目录手册以及使用方法等内容。

经验 13 逐步培养个人的编程思想

提到编程思想，多数程序员都不陌生，可很少有人能说的十分清楚具体。笔者根据多年的编程经验从以下两个方面来谈谈对编程思想的理解。

- 掌握某种编程语言的核心技术，其实每种编程语言都有自己的特色，都有自己解决问题的核心技术，那么这些核心技术就是这种编程语言的灵魂，而其他的扩展技术基本都是在核心技术的基础上发展而来的。
- 分析与解决问题的能力，这里是从编程的角度来说明程序员分析与解决问题的能力，解

决问题能力的高低取决于程序员的专业能力、目标问题领域的专业知识及工作经验等综合因素。编程思想重视的是解决问题的方法论，而编程语言仅是在找到这个问题的解决方案之后的一种实现工具。

编程思想可以通过学习、交流以及在实践中总结出来，掌握了编程思想，就真正领会了这门语言的技术灵魂和掌握解决问题的能力，这就是才是一个合格的程序员。

经验 14　面试经验谈

成功的最佳捷径是让人们清楚地知道，你的成功符合他们的利益。

——拉布吕耶儿

在竞争越来越激烈的今天，如何找到一份工作是当今毕业生最关心的话题，那么又如何在大环境不好、应聘难度增大的情况下脱颖而出呢？如今，找工作就如同是兜售商品，能把自己推销出去才是真正的王者之道。

- 准确定位

首先要定位自己的优势在哪里，自己适合什么样的工作，自己的不足有哪些，根据实际情况明确自己的目标，准确定位才是成功的关键。

- 不打没有准备的仗

需要保持一个良好的心态，一定要放松心情，大方地表现自己，不要唯唯诺诺，以至于说话的声音都有些发抖，这样只会为自己的面试大打折扣，不利于把自己最好的一面展示给面试官。

面试前一定要踏踏实实地准备，不能一个面试电话过来，还不知道对方公司的名字。要对面试公司有个整体的了解，主攻业务是什么？公司文化怎么样？公司的技术又侧重于哪些方面？公司的核心价值观又是什么？做到知此知彼，才能在面试过程中做到收放自如。归纳并总结自己的优缺点，这些可能是面试中经常会问到的问题。

提示： B 君在网上投递简历应聘程序员，但是迟迟没有消息，某日 B 君接到面试公司的电话，一时兴奋过头竟忘记了问招聘公司的名字，去了以后才知道人家招聘的竟是销售人员，令 B 君哭笑不得。

- 衣着大方得体

俗话说“人要衣装，佛要金装”，衣着可以客观地体现出一个人的性格和生活习惯，所以面试尽量不要穿名牌衣服，也不要太随意。应聘者给企业的第一印象十分关键，面试时最重要的是合体、适当、大方，不一定是职业装，但男生一定要穿带领的衣服，不要穿圆领衫，不要穿牛仔裤，皮鞋

一定要干净，头发不要太过新潮，指甲不能长也不能有泥。女同学即使天生丽质，也要化个淡妆，但是绝对不要浓妆艳抹或佩戴大量的首饰。女生如果穿丝袜的话，最好带个备用的，以免丝袜被钩破而又没有备用的可以替换。

提示： C 君在应聘一个公司的计算机工程师时打上了领带，穿上了西装，面试很顺利，技术问题更是对答如流，但是在决定是否录用这名求职者的时候人力资源经理提出了异议：“不知你们注意到没有，这名面试者穿了一双旅游鞋，一方面如此不得体，另一方面也说明他很有个性不好管理”。公司最终放弃录用 C 君。

- 面试时注意自己的肢体语言

谈话时不要有小动作，必须表现出充分的自信，保持正常的坐姿即可。挠头晃腿，摆弄手机，说话的时候挠挠耳朵或不停地搓手，东张西望做小动作……这些小细节都可能会降低你的面试分数。

提示： D 君在面试时坐姿一直保持的都很好，随着面试的深入 D 君也慢慢地放松下来，竟然不知不觉翘起了二郎腿，一会儿又踮起脚来，让面试官非常地反感，并最终选择了别人。

- 坦诚相待

在面试的时候，千万不要刻意地去装饰自己，在面试中坦诚一点，反而会增加自己的分数。而且有些问题我们是无法掩饰的。

提示： E 君为了增加自己面试的成功率，在面试的时候谎称自己在校期间做过 XX 项目，但是当面试官让 E 君介绍一些项目的技术细节时，E 君却支支吾吾，反而起到了适得其反的效果。

我认为，提高自己的核心竞争力才是根本，平时多注重积累一些职业技能，培养自己的沟通能力、领导能力和学习能力。不要害怕失败，“伤痕是一个军人最好的功勋”，失败和挫折才是磨砺我们最好的砥石。只要你在面试中表现出了应有的“成熟”和“专业态度”，那么就不用担心企业会将你拒之门外。



第3章 畅谈编程

想编写出高质量的代码不仅需要深厚的理论知识，还要通过不断的实践来积累经验，成功的道路并不是一帆风顺的，而是布满荆棘与坎坷，想要成就编程高手没有什么秘诀和速成之道，只有付出时间与辛劳不断的学习和积累，当然学习编程也是有法可循、有经验可借鉴的，善于借鉴别人的经验，可以很好地帮助自己缩短到达技术顶端的路程，绕过通往成功的道路障碍。古人早就总结出这个道理，诗经有云："它山之石，可以攻玉"，意思是别的山上的石头，可以取来制作治玉的磨石，引申为"借助外力，改己缺失"。不过有些人很固执，不喜欢向他人虚心请教经验；也有些人爱面子，不好意思向他人请教经验。这些做法都不妥当，因为经验是十分宝贵的，它不仅是成功的总结，更是失败的深刻教训。

今天我们的生活已经与编程密切相关，比如去火车站购票的票务系统、到银行办理存取款业务的金融管理系统、到超市购物的结算系统等，数不胜数，编程给我们的生活带来了方便快捷的同时，也让我们的生活更加丰富多彩，因此我们有理由相信中国软件的明天会更加辉煌。

感悟01 程序员的成长之路

业精于勤荒于嬉，行成于思毁于随。

——韩非子

每个人都是从零开始成长的，程序员当然也不例外。程序员一般会选择走两种路线，一条是技术路线，由程序员，经系统分析师、系统设计师、产品经理，直到架构设计师为归宿，如图3.1所示。这整个过程没个十年半载是办不到的，适用于喜欢技术的人，对技术狂热的人，愿意一辈子去搞技术的人。当然在这其中，需要程序员在工作以及平时的生活中，不断地学习和研究。系统分析师、架构师的工作是令人羡慕的，也是很多公司的挖掘对象。虽然国内的系统分析师和系统架构师比较抢手，但是想真正成为这个角色还是相当困难的，自己必须拥有很强的专业技能和系统分析能力，还有国内对其职位的需求量并不是十分大，只有实力雄厚的公司才会设置系统分析师、架构师的职位，种种原因让这个岗位的人少之又少。

第二种路线就是领导路线，也是大部分程序员的成长路线，当程序员到达高级程序员后，向管理方面进行转变，经开发小组负责人、工程负责人，奋斗到项目经理。这种路线适合有一定沟通能力和协调能力的人。走向管理岗位并不是放弃编程，项目经理会在宏观上调控项目的进行，而不向

程序员那样注重于程序的细节和功能点。

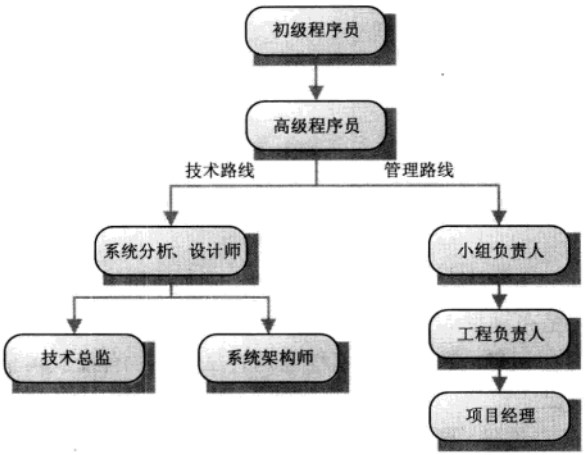


图 3.1　程序员的成长之路

有的人可能会问，程序员人那么多，不可能每个人都能成为系统设计、架构师或是项目经理。这种想法很有道理的，有的人有所成就可能会自立门户，选择自己去当老板，或者去培训中心教书育人，再或者就是程序员的第三种路线，就是告别编程——转行，毕竟在中国不能当一辈子的程序员，这可能是程序员最无奈的一种选择了，但是这却是每个程序员都需要思考的问题。

然而，有一点却是不可否认的，就是成长都要付出艰辛的代价（绝不是混工作经验混出来的），当你成功的时候回过头想想那些付出，就会认为一切都值得。

感悟 02　程序员的生存法则

我们生存在一个竞争的时代，在这个竞争时代对程序员的考验也是严肃的。这里我针对当前形势列举了一些当代程序员的应做到的一些工作细节。

第一条——与时俱进，不断进步。

也是最重要的一条，程序员需要不断的学习来补充自己的能量，因为在 IT 界，技术更新日新月异，如果自己哪天一掉队，想要奋起直追就很困难了，你会发现自己总是别人慢那么一点点。所以程序员的第一条法则就是要跟的上时代和新技术的步伐。所以说程序员都是很聪明的，很少有哪个职业的工作者会像程序员那样掌握一门技术知识那么的全面。

说明：虽然计算机技术更新得很快，但是自己也要有选择地学习，不能出什么新技术自己就扑上去。

第二条——认真负责，少犯错误。

因为程序中一个小小的 Bug 都可能导致系统的瘫痪，所以自己写程序的时候需要不断地进行

测试，减少 Bug 出现的概率，所以程序员都是非常谨慎的。认认真真写程序是程序员的基本生存法则。

第三条——勤奋听话，踏实工作。

现在的项目开发任务都比较繁重，没有正确的工作态度是无法完成的。项目经理喜欢的都是听话的员工，让你做什么就做什么，千万不要自由发挥（自由发挥建立在没有偏离项目方向的基础上），小心偏离了方向，得不偿失。

说明： 项目经理让 A 君做一个简单的可抽拉菜单，但 A 君认为可以做得更漂亮一些，花了大把的时间和精力为菜单加上了一些绚丽的特效，结果验收时菜单和软件总体的稳重风格显的极为不协调，最后 A 君只好把这些费时费力添加的特效去掉。

第四条——注意备份，防患于未然。

不要认为这是无用的，人总是会丢东西的，这是人生的一部分，没有办法避免。说不定哪天你的硬盘就被当掉了，或者你的笔记本电脑被偷了，再或者你的硬盘被病毒格式化了。所以定期（如每天、每周）备份自己的程序和文档是相当必要的，可以免去很多不必要的麻烦。

第五条——掌握多种语言，放宽思维。

不同的程序语言有它适合的不同领域，当你面对某个问题时，需要找一些合适的语言来解决这些问题。花时间去探索其他你所不熟悉的语言，能让你的思路更加广阔，思考的问题也更加全面。

说明： 如果你需要提升程序的性能，可能 C/C++是首选；如果你需要跨平台，那么 Java 才是你需要的编程语言；如果你要开发 Windows 程序，那么.NET 一定是当仁不让的；如果你要写一个 Web 程序，那么你可能会选用 PHP、ASP、JSP 中的一个；如果想开发一个富客户端程序，好像 Flash、JavaFX 和 Silverlight 都是不错的选择。

程序员的的生存法则也自然界中的适者生存的法则是相同的，只有你适应了 IT 这个大环境，你才能在这个行业中占有一席之地。

感悟 03　程序员的生涯规划

近日在网上发现一则小故事很有趣，其实不光是故事中的程序员对自己的未来感到迷茫，有时我们自己可能也会感到迷茫。

一个程序员对自己的未来很迷茫，于是去问上帝：“万能的上帝呀，请你告诉我，我的未来会怎样？”上帝说：“我的孩子，你去问 Lippman，他现在领导的程序员队伍可能是地球上最大的。”而 Lippman 却说：“程序员的未来就是驾驭程序员。”程序员对这个未来并不满意，于是他又去问

上帝："万能的上帝呀，请你告诉我，我的未来会怎样？"上帝又说："我的孩子，你去问 Gates，他现在所拥有的财产可能是地球上最多的。"于是他便去问 Gates，Gates 说："程序员的未来就是榨取程序员。"程序员对这个未来还是不满意，于是他又去问上帝："万能的上帝呀，请你告诉我，我的未来会怎样？"上帝摇摇头："唉，我的孩子，你还是别当程序员了。"

千万不要对自己的前途感到迷茫，无论你是刚踏出校门的大学生还是广大的职场人士，都应该开始考虑对自己进行理性的职业生涯规划与设计。

首先应该制定一个短期目标和长期目标，利用短期目标来牵引和实现长期目标，并实现自己的发展。目标要根据个人的资源、技能和环境配备程度来进行设计，至少是可以实现的目标，不能不合实际。最好再给目标加个时间限制，例如工作 3 年到 5 年自己要成为系统分析或系统设计人员。

说明： 明确了自己的目标以后，最好为目标制定一个规划时间表，每当自己实现了一个短期目标时，就是实现了人生的一个里程碑，长期目标的实现就是我们的成功宣言，它会时时刻刻提醒我们目标是否实现，进展如何，还需要付出多少努力。

然后需要制定出一个明确的实施计划：明确自己要做什么，实现方向究竟如何。例如，你目前已经在一个公司工作，对你来说升职是非常重要的，在这里就需要明确一下怎么做才能符合公司对我的要求和期望，让公司认可你的价值和成绩，让自己在竞争者中脱颖而出。尽量保持自己的贡献在公司的要求之上。

总而言之，每个人至少每隔两年要总结一下自己：对照自己制定的职业规划看看目标达成了多少，这两年是否达到了自己的预期目标，如果没有究竟是环境的原因，还是自己个人的原因，如果是环境的原因，可以考虑是否需要换个环境，如果是自己的原因，可以调整自己或者把目标设的更现实一些。让职业规划的目标给予自己动力，让自己朝着正确的方向发展。

感悟 04 项目开发经验谈

谈起项目开发经验，许多程序员经过几年的实际开发工作都会有所感悟，有的人夸夸其谈项目有多么大、多么重要、都使用了什么高新技术；也有人会谈一些失败和教训，甚者包括埋怨与自责，但我想在大多数程序员内心深处没有表露出的感悟更多的应该是项目开发中经历的痛苦与折磨，穿透这"痛苦与折磨"我们可以品味到那份沉甸甸的成就感和自我价值的实现，但无论成功与失败、高兴与沮丧，经验都是宝贵的，所以我们都要深刻领会。

项目开发经验谈是个比较大的话题，仁者见仁智者见智，内容十分丰富，笔者在这里总结了自己多年的管理软件开发经验及在培训学习中获得的前辈经验，具体如下面论述的若干条。

- 了解行业背景的必要性，在系统调研之前，项目负责人和需求调研分析人员适当了解行业背景是非常必要的，如果你是这方面的行业专家，那就最好不过了，这非常有利于系统调研计划的制定。

- 制定系统调研计划，在做系统调研之前，最好写一份系统调研计划，调研计划编制的水平，取决于项目负责人对被调研行业专业知识的了解程度和项目经验，系统调研计划的内容至少包含以下 3 个方面。
 - 简述调研目的，简要概述本次调研的必要性。
 - 准备调研内容，根据项目负责人对专业知识的了解情况及以往的项目经验列出若干条。
 - 系统调研对象，相关管理人员和系统的使用人员。
- 系统调研要细致，系统调研直接影响需求分析报告的制定，非常重要，系统调研越细致越好，不要怕用户需求多，而且还要善于用启发的方式与用户探讨隐含的或潜在的需求，甚至能够挖掘超出用户期望的令人兴奋的需求最好，当然这需要调研人员尽快完整地熟悉相关业务。几乎没有一个项目能够一次性把需求提完整，都是多次不断地增加、修改，反复研究，所以要求每次调研尽量细致，这样可以减少调研的次数，同时也能够减少需求分析报告和软件架构的修改。
- 尽量与业务负责人沟通，在与用户进行业务沟通时，尽量与业务负责人沟通，不要随便找人就问，因为业务负责人考虑问题相对全面，会考虑问题的得与失，而基层的人员往往考虑自己的方便，而忽略了全局，这一点切忌。
- 重视高层管理者的必要性，在项目开发过程中，有时会遇到违背设计方案的特殊需求，这些需求的加入会严重影响工期，甚至是推翻原有的方案，仅是基层人员的一时之需，并且还可以通过其他变通的方法来解决，但有时基层人员很难说服，这时我们可以向上级主管人员反应情况，说清事实，一般情况下上层主管人员都会通盘考虑问题，给项目组一个比较满意的答复，而且有时管理层级别越高，问题越容易解决。
- 合理控制需求变动，项目中最麻烦的就是需求变动，用户想问题几乎不会按照软件设计框架去想问题，甚至有些变动是翻来覆去的，频繁的变动让项目的原有设计框架变的越来越难以支撑，让项目组成员身心具疲，这时一定要保持良好的心态，然后去寻找解决办法，笔者在项目中总结出以下几种经验供参考。
 - 考虑用变通的方法解决，有些问题直接解决确实非常困难，用一种变通的方法来解决，而且能够得到用户的认可，这会节省许多时间和人力，不过这一点要求项目负责人具有较强的说服力和说服技巧。
 - 登记用户提出的问题，并让相关负责人签字确认，这一点很麻烦，但是效果很好，可以比较有效地防止用户反复修改同一个问题，而且能够让用户在下一次提问题时更加认真考虑。
 - 提前多考虑意外情况，从需求的角度考虑，用户永远没有错，所以在系统调研、需

求分析、软件架构设计中要提前多考虑意外情况，让软件系统能够最大限度地应对各种需求变动。

- 必要的需求变动不要回避，对于用户的需求变动一定要仔细慎重考虑，不要轻易下结论，改与不改。如果用户的需求影响到将来业务的拓展或涉及报表数据等情况时，就必须改动。有时项目组自己也能发现一些业务需求漏洞，对于这些漏洞笔者建议主动提出来与用户进行探讨，不要“隐藏”起来，因为必要的东西，迟早有一天会爆发出来，到时再修改可能难度更大。

- 主动学习业务的必要性，对于这一点许多程序员都有过反感情绪，有人认为做程序员是来编写程序的，而不是来学习行业专业知识的，也有人认为是浪费时间，还不如省出时间来编写程序等许多其他想法，针对这些想法，笔者从以下几方面来论述。

 - 管理软件需要蕴含丰富的管理思想，因为用户要使用软件来辅助人管理企业和提高办公效率，可以形象地说管理软件是有管理思想、有主动管理意识的，一套成功的管理系统是由是50%的业务加上50%的程序开发组成的，若整个项目组都不懂业务，那么项目的研发简直无法想象。

 - 熟悉业务，有利于开发程序，这点做程序员的应该体会非常深刻，如果熟悉业务，写程序会很顺畅，即使遇到比较复杂的业务逻辑，也比较容易编写，相反，不熟悉甚至不懂业务，即使在有编码任务书的情况下，编写程序也十分缓慢或错误不断，所以程序员至少也要熟悉分配给自己的那部分业务。

 - 有利于程序员的职业生涯发展，在众多的程序员中能够登上技术金子塔顶端的人寥寥无几，程序员单凭纯技术来规划自己的职业生涯，并不是最佳的选择，但程序员在工作中接触行业知识的机会比较多，若能够抓住机会比较好地掌握某个行业的业务知识，再结合编程技能，那么你甚至可能成为这个行业的信息技术专家，而这种业务与技能相结合的复合型技术人才正是社会所需的高端人才，如 ERP 顾问等。

说明： 当然您可能认为还有其他更重要的情况，但限于篇幅的原因，笔者基于自己的经验仅列举这些供大家参考。

感悟 05 信息技术改变我们的生活

可以说信息技术是对现代生活影响最大的因素之一，它已经渗透到我们生活的各个角落，甚至连战争都提倡信息化。信息技术改变了我们生活和工作的方式，它让我们生活的更加丰富多彩，让我们工作更方便，极大地提高了人类进步的速度。笔者从以下几个方面来论述信息技术对我们生活的改变。

- 商务电子化，电子商务是信息技术应用的重要方面之一，它代表着未来贸易方式的发展

方向。由于计算机互联网的普及，使全球经济一体化变得很容易，大量的商品、服务、技术等各种信息都在全球范围内广发流动。世界上的任何企业或个人，只要登录因特网，都可以随时、随地地提供或获取相关信息，并可以迅速完成交易。电子商务交易成本极低，从而使企业或个人得到最大的经济效益。电子商务离我们并不遥远，有许多人接触过，比如国内比较有名的淘宝网、阿里巴巴网、京东商城等电子商务交易平台。

- 政务信息化，政务信息化有利于提高政府的行政效率，推动机构改革，使政务工作公开化、透明化，并规范了行政工作的执行。这些年我国政府信息化建设非常迅速，几乎各级政府都有自己的政务网站，通过网站，让老百姓对政府的工作有了更加深刻的了解，同时，政务信息化能为公众提供更有效的服务，凡是公众与政府发生的接口，都可以考虑利用计算机网络来改进服务，方便公众。

- 教育电子化，教育电子化丰富了原有的教育方式，可以通过多媒体教学，让教学形象生动，不枯燥；可以通过远程教育，达到合理配置教育资源的目的。

- 生活便捷化，这方面体会就更加深刻了，如网上聊天、网上查资料、网上购物等。

- 娱乐信息化，最具代码表性的莫过于网络游戏和动漫，它们丰富了我们的业余生活，更是国民经济的一个重要产业，它相对重工业能源消耗小、污染小，但利润却很高，同时带动了许多相关联的其他产业，相关联的产业又带动了人员的就业。

- 医疗信息化，医疗信息化这些年比较显著的成就是远程医疗技术的应用，远程医疗的实现，为医生和患者同时提供了方便，客服了因地域、医疗设备等造成的诸多不便，并对平衡不同地区之间的医疗技术差异起了重要作用。

2

第 2 篇
陷阱与谬误
——汲取前人的教训

第 4 章　开发人员不可忽略的 22 个技术陷阱
第 5 章　36 个开发人员意识中的常见谬误



第4章　开发人员不可忽略的22个技术陷阱

在编写程序的过程中，难免会出现错误。但面对错误，要冷静分析，保持思维的条理清晰。有的错误让程序员感觉不可思议，甚至深陷其中而不能自拔。笔者写程序多年，也曾出错无数，现将开发项目过程中遇到的陷阱进行总结和概括，列出22个典型的陷阱供读者朋友们学习。希望读者朋友们阅读本章的内容之后不再犯同样的错误。本章在分析陷阱、解决陷阱的过程中融入了笔者的分析思路和排错技巧，对调试技巧感兴趣的读者朋友们可以细心去体会。

陷阱01　忽略了大小写的区别

1. 陷阱产生的场景

C语言中的标识符是区分大小写的。编译程序把a和A认为是两个不同的变量名。而有些初学者常常会忽视这个问题，而造成程序的编译错误。

例4.1　忽略标识符大小写产生的陷阱（光盘位置：光盘\MR\Instance\4\01），本实例实现定义一个变量a，对变量进行运输，并输入变量最后的值。该实例的代码如下：

```
#include <stdio.h>
void main()
{
    int a=10;                //定义变量
    a+=a;                    //变量值加一
    printf("%d\n",A);        //输出
}
```

运行程序，却出现了如图4.1所示的错误信息。



图4.1　编译错误信息

2. 陷阱的分析

C语言是区分大小写的，上面实例代码中定义的的是小写字母的变量a，然而在输出时却由于疏忽写成了大写的字母A，由于这个变量没有定义，所以编译器提示了A没有被定义的错误信息。

说明：一般习惯上将变量定义为小写，常量定义为大写，以增加程序的可读性。

3. 陷阱的解决方法

将输出语句中的 A 改为小写的 a。代码如下所示：

```
printf("%d\n",a);
```

程序运行效果如图 4.2 所示。

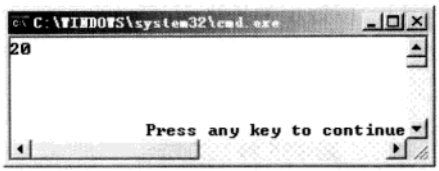


图 4.2　程序运行窗口

陷阱 02　“{}”与“()”使用不当造成错误

1. 陷阱产生的场景

有些初学者对于“{}”和“()”的使用容易混乱，不知何时使用大括号，何时使用小括号。有些专业的程序与有时也会由于疏忽错用了这两个括号，从而造成程序执行结果的不正确。看一下下面的例子。

例 4.2　在程序中初始化一个二维数组，并输出数组元素（光盘位置：光盘\MR\Instance\4\02），该实例的代码如下：

```
#include <stdio.h>
void main()
{
  int i,j;                                    //声明变量
  int a[2][3]={(1,2,3),(4,5,6)};              //定义数组
  printf("array a :\n");                      //输出
  for(i=0;i<=1;i++)                           //循环两次
  {
    for(j=0;j<=2;j++)                         //循环三次
    {
        printf("%5d",a[i][j]);                //输出数组
    }
    printf("\n");                             //输出回车
    }
}
```

本实例的目的是初始化一个二维数组，并将二维数组中的数据输出。不仔细看觉得没什么错误，但运行结果却出人意料，程序运行结果如图 4.3 所示。

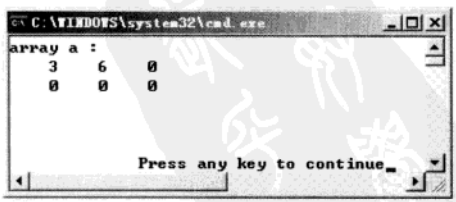


图 4.3　输出二维数组中的数据

2. 陷阱的分析

从执行结果中，可以得出这样的结论，数组中只有前两个元素被赋值，而且不是正确的，其他元素都没有赋值。

仔细查看代码发现，代码中在初始化二维数组的时候将二维数组的“{}”误写成了“()”。这样在初始化赋值的时候执行的是括号运算，只得到了 3 和 6 两个值。其他数组元素并没有被赋值。所以产生了如图 4.3 所示的运行结果。

二维数组初始化的时候可以使用按行赋初值，即将每行的元素使用花括号“{}”括起来的。这样的赋值方法比较直观，也经常被使用。但是有些初学者由于疏忽会将花括号“{}”写成了小括号“()”，这样程序编译不会产生错误，但是也不会被正确赋值。

3. 陷阱的解决方法

改正后的主要代码如下：

```
int i,j;
int a[2][3]={{1,2,3},{4,5,6}};                    //初始化数组
```

程序运行效果如图 4.4 所示。

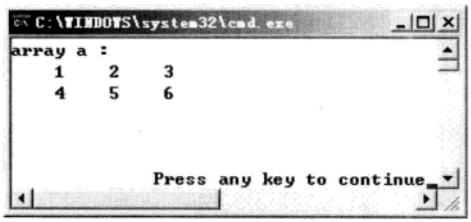


图 4.4 输出二维数组中的数据

陷阱 03 多加分号导致出错

1. 陷阱产生的场景

在 C 语言中分号“;”用于结束一个语句，就如同平日常用的句号“。”作用一样，但是并不是每一条的 C 语言语句都需要在后面加上分号作为结束标志的。if 和 while（表达式）等的后面就并不需要加上分号，如果在编写程序的时候，不小心加上分号，大多数的情况编译器就会报错。但有时也会是程序产生意想不到的错误。

例 4.3 在 while 语句后使用了分号产生死循环（光盘位置：光盘\MR\Instance\4\03），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
    int x = 10;                                   //定义变量
    int sum = 0;                                  //定义求和变量
    while( x > 0 );                               //循环
   {
     x--;                                         //计数变量自减
     sum=sum+x;                                   //进行运输
   }
   printf("%d",sum);                              //输出
}
```

这里使用 while 语句进行循环，要计算从 10 到 1 的累加结果，然后由于疏忽产生死循环，以至于程序在不断的执行，不能输出结果。运行效果如图 4.5 所示。

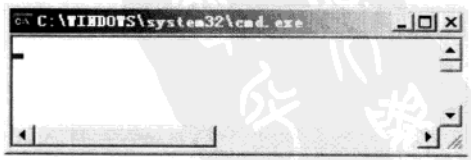


图 4.5 产生死循环得不到结果

2. 陷阱的分析

C 语言中并不是所有的语句后面都要加上分号，例如定义宏时的 define 语句、引用文件时的 include 语句或者是 if 和 while 以及 for 的表达式等，都不用加分号。上面的实例在 while 语句之后加上了分号，就表示 while 后面跟着一个空语句，并没有对计数变量进行操作，所以产生了死循环。

3. 陷阱的解决方法

将 while 语句后的分号去掉。

注意：本实例在光盘中给出的源代码即为死循环情况，请读者使用时注意。

程序正确的运行结果如图 4.6 所示。

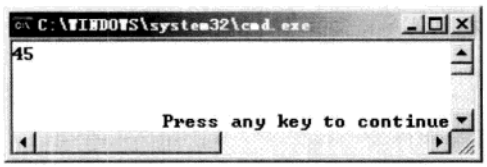


图 4.6　程序预期运行结果

陷阱 04　使用循环语句不慎产生死循环

1. 陷阱产生的场景

在 C 语言中常用 3 种循环语句，这些循环语句各有其特点，while 和 do…while 经常用在循环次数不确定的场合；for 被经常用来遍历数组和集合。在使用这些循环语句过程中，若其循环结束条件处理不当，则很可能产生死循环，本实例对循环语句出现的死循环情况进行详细分析。

例 4.4　使用循环语句不慎产生死循环（光盘位置：光盘\MR\Instance\4\04），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
    unsigned int i;                          //定义无符号整型变量
    for (i=10;i>=0;i--)
    {
        printf ("%d ",i);                    //输出
    }
    printf("\n");                            //输出回车
}
```

这里是想输出每次得到的计数变量的值，结果由于疏忽导致程序死循环。运行效果如图 4.7 所示。

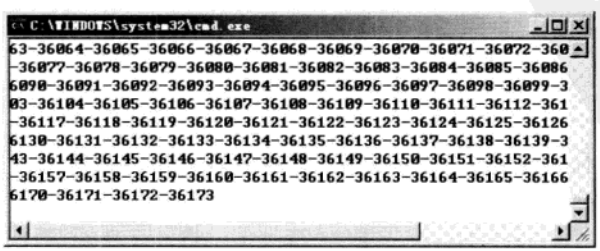


图 4.7　产生死循环程序永不结束

提示： 上面的方法在执行时产生死循环。在实际的应用程序开发中，产生死循环的情况比较复杂，并不完全像本实例中的情况，但归根结底都是由循环结束条件处理不当造成的，下面将对这些方法进行详细地分析。

2. 陷阱的分析

for 语句的循环结束条件是当条件表达式的值为 false 时。上面实例中的计数变量 i 被定义为 unsigned 类型，这样 i>=0 就永远成立。所以程序进入了死循环。

下面为另外两种常用的循环语句的判断条件。

- while 语句的循环结束条件是当布尔表达式的值为 false 时。
- do…while 语句的循环结束条件是当布尔表达式的值为 false 时。

注意： 使用 do…while 语句时要注意循环次数的判断，因为 do…while 语句是先执行循环体后进行判断。

3. 陷阱的解决方法

将计数变量 i 定义为 int 型。这样就不会出错了。程序的正确运行效果如图 4.8 所示。

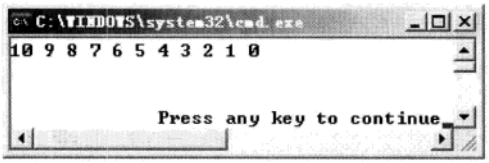


图 4.8　程序运行效果

注意： 本实例在光盘中给出的源代码均为死循环情况，请读者使用时注意。

陷阱 05　忽视 do...while 循环次数产生的错误

1. 陷阱产生的场景

前面实例中讲到 C 语言中经常用到的循环语句有 for 循环、while 循环和 do…while 循环。其中使用 while 和 do…while 循环的时候要注意两者之间的细微差别，do…while 循环在执行的时候总是要先执行循环体，然后判断条件是否成立；而 while 则先执行判断。在编写程序的时候要注意，以免由于多执行了一次循环而产生结果不正确的错误。

例 4.5　忽略 do…while 与 while 的区别，使用 do…while 循环产生结果不正确的错误。（光盘

位置：光盘\MR\Instance\4\05）。

使用 do...while 循环和 while 循环的代码分别如下：

```
#include <stdio.h>
void main()
{
   int i, sum =0;
   scanf("%d",&i);
  do
   {
     sum=sum+1;
     i++;
   }  while(i<=10);
   printf("sum=%d\n",sum);
}
```

```
#include <stdio.h>
void main()
{
        int i, sum =0;          //定义变量
        scanf("%d",&i);         //输入整数
        while(i<=10)
        {
     sum=sum+1;                 //值加一
          i++;
        }
   printf("sum=%d\n",sum);      //输出
}
```

分别运行这两个程序，输入 1，上面两个实例得到的结果都为 10。但是当输入 11 时，do...while 循环得到的结果为 1，while 循环得到的结果为 0。

2. 陷阱的分析

在前面已经介绍 do...while 循环需要首先执行一次循环体然后再进行条件判断，而 while 则直接进行条件判断。这样当循环判断条件为假（0 值）是，do...while 先执行一遍循环体，然后跳出，while 则不执行循环体。所以 do...while 例子中的 sum 就被加了一个 1。

提示： 当 do…while 和 while 的判断表达式相同时。判断时当第一次的值为真时，两种循环得到的结果相同。否则 do…while 循环要多执行一次，使得到的结果不同。

3. 陷阱的解决方法

在进行代码编写时，要注意 do...while 和 while 的差别。根据实际情况选择该使用哪种循环。

陷阱 06　数据溢出产生的陷阱

1. 陷阱产生的场景

C 语言数值数据类型都有一个取值范围。在进行数值运算的时候，有时候会因数值超出了定义类型的取值范围而产生溢出的情况。

例 4.6　整形数据的溢出（光盘位置：光盘\MR\Instance\4\06），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
    int i,rst;                  //声明变量
    i=32767;                    //定义变量
    rst=i+1;                    //变量值加一
```

```
    printf("%d,%d",i,rst);          //输出结果
}
```

在上面的代码中，由于计算得到的数值大于定义的数据类型的范围，所以运行得到的结果不是预期的结果。运行结果如下图 4.9 所示。

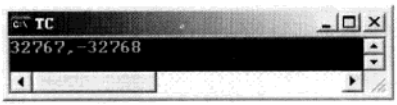


图 4.9　运行结果

2. 陷阱的分析

在 C 语言中可以使用各种类型的标识符。但是 C 语言标准里并没有规定这些类型的具体长度，要由各个 C 编译系统自己规定。一般以一个机器字存放一个 int 数据，早期的计算机字长一般为 16 位，故以 16 位存放一个整数，整数的范围在−327 68～32 767 之间，整数范围太小，在使用的时候容易产生溢出现象。但现代的计算机一般都为 32 位以上，以 32 位存放一个整数，范围可达正负 21 亿，所以当前一般将 int 和 long 都定义为 32 位。在写程序时要了解所用系统对标识符长度的规定，以免出现上面的错误。本程序使用 Toubo C 进行编译，在 Toubo C 中一个 int 型变量的最大允许值为 32 767，如果再加 1 则“溢出”。

如果真的产生“溢出”，运行时也不会报错。而是将结果从“头”开始计算。即是上面的运行结果−32768。这与编程者的原意不同。由于系统不会给出“出错信息”，所以要靠程序员的细心和经验来进行排错。

说明： 当前大多编译器多将 int 规定为 32 位，所以一般不会出现“溢出”现象。

3. 陷阱的解决方法

将数据定义为长整型 long 类型。在输出的时候使用“%ld”即可。

提示： Long 型数据可以得到大范围的整数，但同时会降低运算速度，因此除非不得已，否则不要随便使用 long 型。

陷阱 07　整数的除法导致结果的偏差

1. 陷阱产生的场景

C 语言中的除法运算符“/”有着比较灵活的规定。两个整数相除结果为整数，运算中的两个数有一个数为实数，则结果是 double 类型，因为所有实数都按照 double 型进行运算。做整数除法的时候舍去小数部分，但是如果除数或者被除数中有一个为负值，则舍入的方向是不固定的，多数机器采取“向零取整”的方式进行运算。如 5/3=1，−5/3=−1，向零取整。

例 4.7　整数的除法导致不是预期的结果（光盘位置：光盘\MR\Instance\4\07），本实例目的是

想对实数 i 进行数学运算得到一个结果。主要代码如下：

```
#include <stdio.h>
void main()
{
    double i;                    //声明 double 型变量
    i=9.5+1/2;                   //为变量赋值
    printf("%f",i);              //输出结果
}
```

这里是进行实数运算，是想要将 9.5 与 1/2 计算的结果 0.5 相加，得到数值 10。然后事与愿违，运行效果如图 4.10 所示。本例的运算结果为 9.5。

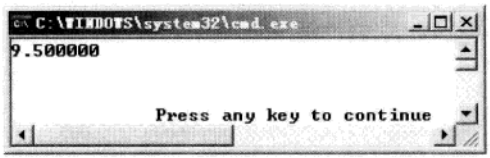


图 4.10　运行结果

2. 陷阱的分析

使用除法运算符“/”对两个整数进行运算的时候，得到的结果也是一个整数。1/2 得到的结果去掉小数部分得到的整数为 0。所以上面的运算结果不是预期的结果。

3. 陷阱的解决方法

将 1/2 改为 1.0/2 即可得到一个实数的结果。将除数和被除数中的任意一个数设置为实型，都会得到一个双精度的结果。代码如下：

```
double i;
i=9.5+1.0/2;
printf("%f",i);
```

陷阱 08　变量未定义

1. 陷阱产生的场景

变量未定义也是经常出现的小错误，大多是由于编程者疏忽所致。

例 4.8　变量未定义产生的编译错误（光盘位置：光盘\MR\Instance\4\08），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
    int x = 10;                  //定义变量
    while( x > 0 )               //循环
    {
        x--;                     //自减
        sum=sum+x;               //加法运算
    }
    printf("%d",sum);            //输出
}
```

运行程序执行上面的代码，会提示一个编译错误。如图 4.11 所示。告诉用户，此变量没有定义。

2. 陷阱的分析

变量是用于存储数据的。一个变量都有一个名字，在内存中占据一定的存储单元，在该存储单元中存放变量的值。在 C 语言中，要求对使用的变量做定义，就是要先定义，后使用。这样能够方便确定变量类型并为其分配存储单元，同时也便于在编译时检查该变量进行的运算是否合法。使用未被定义的变量，在编译时就是提示错误信息。

3. 陷阱的解决方法

将未定义的变量进行定义。如下代码：

```
int sum = 0;
int x = 10;
```

程序正确的运行效果如图 4.12 所示。

```
error C2065: 'sum' : undeclared identifier
```

图 4.11　编译错误

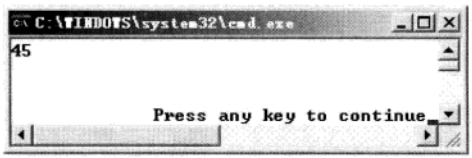


图 4.12　程序运行效果

说明：变量未定义的错误在程序编译时会提示错误信息。这种错误很容易被发现。

陷阱 09　输入数据时忘记添加&符号

1. 陷阱产生的场景

C 语言中使用 scanf 函数可以按照指定的地址将程序运行时键盘输入的数存入内存。但是有时由于编程者疏忽会无法接收键盘输入值，例如下面的例子。

例 4.9　输入数据到变量时忘记在变量前添加&符号（光盘位置：光盘\MR\Instance\4\09），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
    int a,b;                          //声明变量
    printf("输入两个整数\n");
    scanf("%d%d",a,b);                //输入
    printf("%d,%d\n",a,b);            //输出

}
```

上面的代码是想要在键盘上输入两个数值，并分别存入变量 a、b，但是由于疏忽没有得到输

入的值。程序出现异常错误，提示如图 4.13 所示错误信息。

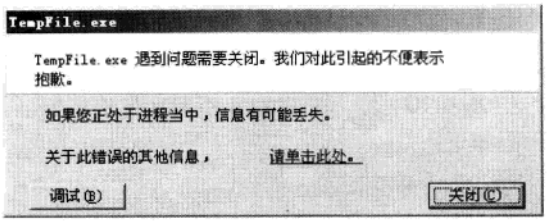


图 4.13　异常提示

2. 陷阱的分析

scanf 用于按照指定的内存地址将键盘输入的数据值存到相应地址中。其一般形式为：

```
scanf(格式控制，地址列表)
```

如：

```
scanf("%d%d",&a,&b);
```

- “格式控制”是由双引号引起来的字符串，该字符串由格式说明和普通字符组成。
 - 格式说明由“%”和格式字符组成，如%d、%f 等。它的作用是将输入的数据转换为指定的格式，格式说明总是由“%”开始的。
 - 普通字符，需要直接输出的字符。例如逗号、空格和换行符等。

说明：“格式控制”可以不用使用字符分隔，例如直接写成“%d%d”。在输入数据时，两个数据直接以一个或多个空格间隔，也可以使用<Enter>键或<Tab>键。但是使用“%d%d”格式时，输入数据不能用逗号作为分隔。

- “地址列表”是由若干个地址组成的列表，可以使变量的地址，也可以是字符串的首地址。

注意：地址列表中应用的是变量地址，而不应该是变量名。许多初学者常常在此出错，希望在使用的时候多加注意。

3. 陷阱的解决方法

根据上面的分析可知，要将输出数据存入变量，要使用变量地址，即在变量前加上地址符。如下代码：

```
scanf("%d%d",&a,&b);
```

程序的正确运行效果如图 4.14 所示。

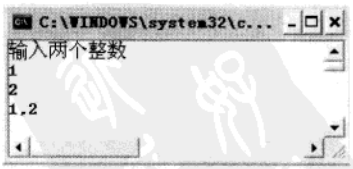


图 4.14　程序运行效果

陷阱 10　使用 scanf 函数输入格式不符

1. 陷阱产生的场景

在上面的错误陷阱中介绍了使用 scanf 函数输入数据由于地址列表错误而引发的程序异常。在

使用 scanf 函数进行输入数据时有严格的格式要求，如果没有按照设置的“格式控制”进行输入就会产生错误。

例 4.10 输入数据格式不符产生的陷阱（光盘位置：光盘\MR\Instance\4\10），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
  int a,b;                          //声明变量
  scanf("%d,%d",&a,&b);             //输入两个整型数
  printf("%d,%d",a,b);              //在屏幕上输出
}
```

运行以上代码，在输出窗口在键盘上输入“12 34”，12 和 34 中间输入了一个空格。按回车键得到运行结果如图 4.15 所示。输出的结果并不是刚刚输入的数据，产生了错误。

2. 陷阱的分析

使用 scanf 函数输入数据时，要求在程序运行窗口输入的数据跟“格式控制”的格式相同。格式控制中没添加任何符号则可以使用空格、<Enter>键或<Tab>键进行间隔；格式控制中使用了逗号或分号等符号，则在输入的时候就要是用这些符号进行分隔数据，否则就会输入错误的数据。

分析代码，在使用 scanf 函数输入数据时，用的“格式控制”是“%d，%d”，中间用了一个逗号，所以在程序运行窗口输入数据的时候，也要在两个数据中间加一个逗号，这样才能够输入正确。

注意：使用错误的格式控制输入数据并不会产生一个错误，只是没有输入正确的数据。这样的错误需要自己发现，只是使得到的结果不是编程者所要的。

3. 陷阱的解决方法

按照“格式控制”输入数据，程序中“格式控制”中使用了逗号，所以在输入数据的时候也要使用逗号。如图 4.16 所示，才能得到正确的数据。

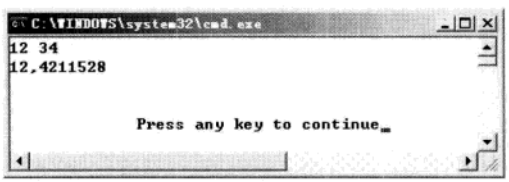


图 4.15 运行窗口

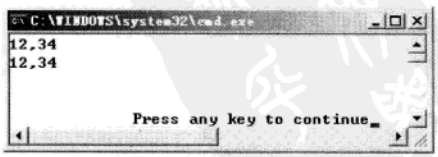


图 4.16 正确的输入

注意：在输入的时候使用的逗号样式也要与“格式控制”中相同，“格式控制”中使用的是英文样式的逗号，在输入的时候就要使用英文样式的，否则也不会得到正确的结果。

陷阱 11　输入的数据类型不符的陷阱

1. 陷阱产生的场景

使用 scanf 函数进行输入数据时，函数中的“格式控制”包含了对要输入数据的格式说明。由“%”和格式字符组成，如%d、%f 等。它的作用是将输入的数据转换为指定的格式。所以在输入数据时要根据给定的格式进行输入，否则将不会得到预期的效果。

例 4.11　输入的数据格式不符的陷阱（光盘位置：光盘\MR\Instance\4\11），该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
  int a,b;                                    //声明变量
  scanf("%d%d",&a,&b);                        //输入数据
  printf("%d,%d\n",a,b);                      //输出
}
```

如上面的代码，是想在程序运行时输入两个数值，然后再将输入的数值输出。在程序运行时，输入两个英文字符 s 和 t，按下<Enter>键，得到如图 4.17 所示的结果。它显然不是预期的效果。

2. 陷阱的分析

分析代码，将输入变量设置成了整型，而输入的是字符型数据，所以在运行时显示的结果不正确。

提示： C 语言的输入输出操作是最基本的操作，而其格式要求比较繁琐，使用不对就不会得到预期的结果。因此读者需要掌握输入输出知识细节，以免在使用的时候由于此方面错误而浪费大量时间。

3. 陷阱的解决方法

如果要输入的是字符型数据，则要将输入变量设置为字符型，并且在使用 scanf 函数的时候格式控制符应写为“%c”，修改后的代码如下：

```
#include <stdio.h>
void main()
{
  char a,b;
  scanf("%c%c",&a,&b);
  printf("%c,%c\n",a,b);
}
```

程序的正确运行效果如图 4.18 所示。

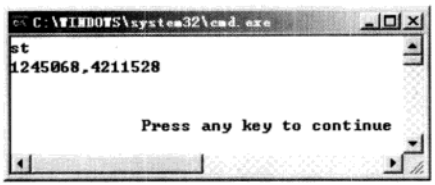


图 4.17 空引用异常提示

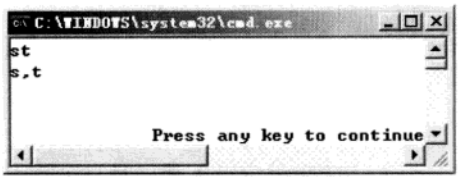


图 4.18 程序运行效果

注意：在使用“%c”格式输入字符时，空格也算作一个有效的字符输入，而不能使用空格作为间隔符了。

陷阱 12 switch 语句中忘记使用 break

1. 陷阱产生的场景

C 语言中的 switch 语句是多分支选择语句，使用 case 语句描述每个分支。当表达式的值与某个 case 后面的常量表达式的值相等时，就执行此 case 后面的语句，需要使用 break 语句跳出 switch，否则程序会在执行的 case 语句之后继续执行下面的代码。例如，有 1～6 号 6 个门，3 号门符合执行任务的条件，程序会直接进入到 3 号门执行任务，如果在任务执行后使用了 break，则跳出，此次任务执行完成；如果 3 号门没有添加 break 跳出命令，程序从 3 号门出来，还会去 4 号门执行任务，直到遇到 break 或者将后面 case 的都执行完。做了无用功不说，还会产生一些不必要的麻烦。

但是有些初学者常常忘记在 case 语句结束后添加 break，而产生错误。

例 4.12 switch 语句中忘记使用 break 产生的陷阱（光盘位置：光盘\MR\Instance\4\012），本实例预期实现运行程序后，输入表示分数段的字符，将输入该字符代表的分数段。该实例的主要代码如下：

```
#include <stdio.h>
void main()
{
 char grade;                          //声明变量
 scanf("%c",&grade);                  //输入字符型数据
 switch(grade)                        //选择语句
 {
    case 'A':printf("90~100\n");      //条件分支
    case 'B':printf("75~89\n");
    case 'C':printf("60~74\n");
    case 'D':printf("<60\n");
    default:printf("error\n");
    }
}
```

运行程序后，输入一个大写字母 B，按下<Enter>键后，程序运行效果如图 4.19 所示。

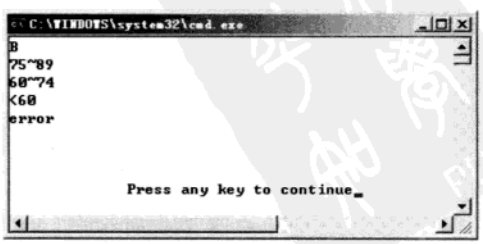


图 4.19 运行结果

2. 陷阱的分析

从上面运行结果看出，程序首先得到了预期的结果，即输入的字母 B 对应的分数段，但是 B 后面对应的分支也都被执行显示出来，程序没有在执行 B 分支后跳出 switch。所以判断缺少了跳出语句 break。

3. 陷阱的解决方法

在每个 case 语句对应的代码后面添加 break，用于跳出 switch。代码如下：

```
switch(grade)
 {
    case 'A':printf("90~100\n");break;                    //带 break 的条件分支
    case 'B':printf("75~89\n");break;
    case 'C':printf("60~74\n");break;
    case 'D':printf("<60\n");break;
    default:printf("error\n");
    }
```

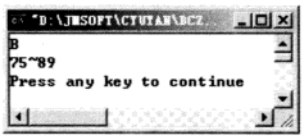


图 4.20　运行结果

程序正确运行效果如图 4.20 所示。

陷阱 13　指针没被初始化

1. 陷阱产生的场景

大家都知道，指针是专门用来存放地址的，被定义为“指针类型”。指针需要指向一个变量才具有实际意义。而有些初学者会由于疏忽使用了没有被初始化的指针，而产生了错误。

例 4.13　指针没被初始化而产生的错误（光盘位置：光盘\MR\Instance\4\13），代码如下：

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *str;                    //声明指定变量
    strcpy(str, "good");          //复制字符到指针变量
    printf("%s",str);             //输出
    return 0;
}
```

本程序的功能是将一个字符串复制到一个指针变量所指向的地址中。然后输出这个指针变量指向内容的值。运行上面的代码，编译器提示一个警告信息，提示指针没有被初始化。如图 4.21 所示。

```
warning C4700: local variable 'str' used without having been initialized
```

图 4.21　编译错误提示

2. 陷阱的分析

指针变量只能存放一个地址，没有初始化的指针变量是没有意义的。在使用一个指针的时候，

首先要思考的问题就是这个指针指向什么变量，如果指针变量没有指向任何变量，程序就是有错误的，这种类型的错误常常导致 UNIX/Linux 上的 fault/coredump 错误或者 Windows 下的一个常规保护错误，一般编译器会提示一个警告信息（而在 Dos 下任何事都可能发生）。

3. 陷阱的解决方法

在定义了指针变量后，要对其进行初始化。例如使本实例中的指针指向一个已分配的字符串数组空间，代码如下：

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *str=malloc(10);              //初始化指针
    strcpy(str, "good");               //复制
    printf("%s",str);                  //输出
    free(str);                         //释放空间
    return 0;
}
```

程序正确的运行效果如图 4.22 所示。

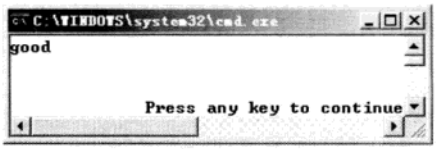


图 4.22 程序运行效果

陷阱 14 操作超出动态分配的内存

1. 陷阱产生的场景

使用 malloc 函数可以在内存的动态存储区分配指定长度的连续空间。此函数返回值是一个指向分配域起始地址的指针，类型为 void。

例 4.14 操作超出动态分配的内存引起的异常（光盘位置：光盘\MR\Instance\4\14）。本实例要实现使用一个指针变量指向一个动态分配的地址。为指针变量指向的地址复制一个字符串。然后将该字符串输出，该实例代码如下：

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *str=malloc(5);    //指针指向动态分配的空间
    strcpy(str, "goodluck");//将字符串复制到指针指向
的空间
    printf("%s",str);       //输出字符串
    free(str);              //释放空间
    return 0;
}
```

程序运行后出现如图 4.23 所示的异常提示。

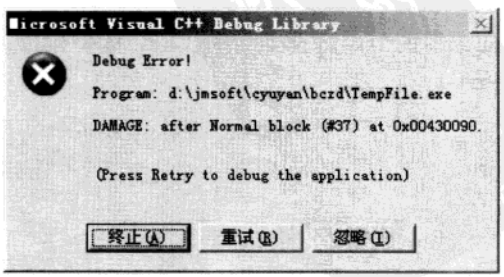


图 4.23 内存使用异常

2. 陷阱的分析

使用 malloc 函数分配内存空间时需要指定需要

开辟的空间的大小，其调用形式为：

```
malloc(size)
```

size 是字节数，需要事先了解要申请空间的大小。如果在操作时，超出了这个分配的空间，则会产生一个异常。

3. 陷阱的解决方法

将申请分配空间的值改的大一些，修改后的关键代码如下：

```
char *str=malloc(20);
```

程序正确的运行效果如图 4.24 所示。

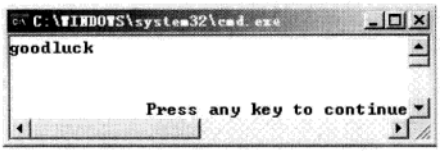


图 4.24　程序运行效果

陷阱 15　使用内存后未释放

1. 陷阱产生的场景

使用 malloc 或者 calloc 函数在内存的动态存储区中为变量分配空间，在使用之后要使用 free 函数，释放这部分内存区，使其能被其他变量所使用。否则会由于内存过多的占用造成的内存空间不足。

例 4.15　使用内存后未释放的陷阱（光盘位置：光盘\MR\Instance\4\15），该实例的实现代码如下：

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *str=malloc(20);                    //指针指向动态分配的空间
    strcpy(str, "goodluck");                 //将字符串复制到指针指向的空间
    printf("%s",str);                        //输出
    return 0;
}
```

说明：内存未释放在程序运行时，开始不会出现错误。当过多内存被占用未释放的时候会造成内存空间不足而死机的现象。

2. 陷阱的分析

使用 malloc 函数或者 calloc 函数分配内存后，这块内存就被占用，直到被释放才能被其他变量使用。所以动态分配内存使用完成后要进行释放，否则如果占用的内存过多当内存不足时，会造成死机的状况。

3. 陷阱的解决方法

使用 malloc 函数或者 calloc 函数分配的内存要使用 free 函数进行释放。其函数原型为：

```
void free(void *p);
```

p 是指向内存区的指针。程序运行效果如图 4.25 所示。

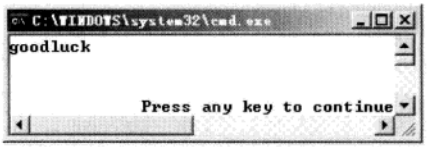


图 4.25 程序运行效果

陷阱 16 内存释放了继续使用

1. 陷阱产生的场景

上面的例子中介绍了使用 malloc 或者 calloc 函数动态分配内存后，在使用之后要使用 free 函数，释放这部分内存区，使其能被其他变量使用。内存释放后，改内存就不能再被使用。

例 4.16 内存释放后继续使用，产生的错误（光盘位置：光盘\MR\Instance\4\16），本实例实现为指针变量动态分配内存，将一个字符串复制到内存中，然后使用指向该内存的指针对该内存进行操作。代码如下：

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *str=malloc(20);                    //指针指向动态分配的空间
    strcpy(str, "goodluck");                 //复制字符串到指针指向的位置
   free(str);                                //释放空间
    printf("%s",str);                        //输出
    return 0;
}
```

运行程序，如图 4.26 所示。

2. 陷阱的分析

分析上面的代码得知，str 是指向动态分配的内存，使用字符串复制函数在该内存复制了一个字符串后，将该内存释放。然后下面仍读取该内存中的数据所以输出的结果不正确。

3. 陷阱的解决方法

在内存使用结束之后再释放该内存。将上面的代码修改为：

```
char *str=malloc(20);
strcpy(str, "goodluck");
printf("%s",str);
free(str);
return 0;
```

先使用该内存，使用完成后再释放即可。程序运行结果如图 4.27 所示。

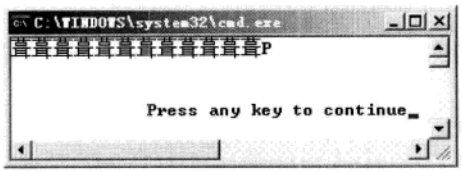


图 4.26　警告信息

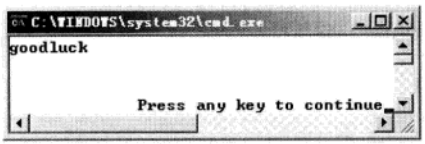


图 4.27　程序正确运行结果

陷阱 17　没有说明函数的头文件

1. 陷阱产生的场景

程序中经常会用到一些库函数或者是包含在其他文件中的函数，使用这些函数的时候，要在代码的开头处使用#include 命令进行说明，将包含函数的文件包含到程序文件中。

例 4.17　没有说明函数的头文件产生的错误（光盘位置：光盘\MR\Instance\4\17），本实例的代码如下：

```
#include <stdio.h>
void main()
{
    int n,i,k;
    scanf("%d",&n);                          //输入一个整数
    k=sqrt(n);                               //求平方根运算
    for (i=2;i<=k;i++)
    {
        if(n%i==0)                           //如果不能整除则跳出本次循环
            break;
    }
    if(i>k)                                  //输出结果
        printf("%d is a prime number\n",n);
    else
        printf("%d is not a prime number\n",n);
}
```

编译程序，提示警告信息，如图 4.28 所示。

```
warning C4013: 'sqrt' undefined; assuming extern returning int
```

图 4.28　警告信息

警告信息提示“sqrt”没有被定义。

2. 陷阱的分析

在程序中使用库函数或者是其他文件中的函数，要在代码开头也就是程序文件头部进行说明。这叫“文件包含”处理，将一个源文件的全部内容包含到另外一个文件中，这种文件被称为头文件。本实例中 sqrt 函数属于库函数 math.h 文件中的函数，所以在使用此函数的时候，要在代码的开头将 math.h 文件包含进来。

文件包含操作的一般形式为：

```
#include<文件名>
```

或

```
#include"文件名"
```

注意：#include 语句后面不加分号。

以上两种写法都是合法的。习惯上规定使用尖括号（即<文件名>）时，系统到存放 C 库函数头文件所在的目录中寻找要包含的文件，是标准方式。用双引号（即“文件名”）形式时，系统现在用户当前目录中寻找要包含的文件，若找不到，再按照标准方式查找。也就是说如果调用的是库函数则使用尖括号，如果调用用户自己编写的文件中的函数，则使用双引号。这样能够节省查找时间，也可以通过符号辨认文件来源。

说明：头文件除了可以包含函数原型和宏定义以外，也可以包含结构体类型定义和全局变量等。

3. 陷阱的解决方法

在代码头部说明包含 sqrt 函数的头文件 math.h，代码如下：

```
#include<math.h>
```

程序的运行结果如图 4.29 所示。

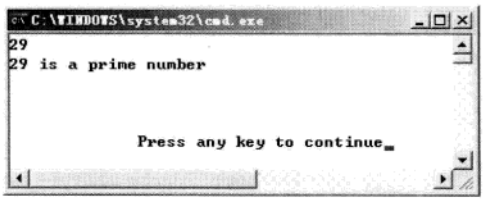


图 4.29 程序运行结果

陷阱 18 忽略了字符串结束符

1. 陷阱产生的场景

C 语言为了判定字符串是否结束，会自动的在每个字符串的结尾加上一个字符串“结束标志”“\0”，系统会根据此标志来判断字符串是否结束。在使用字符数组操作字符串的时候，可以不考虑结束符“\0”。但是如果为字符串动态分配空间，则要为这个结束符也保留空间，否则在存储的时候会超出内存边界，引起内存使用的异常。

例 4.18 动态分配字符串空间忽略字符串结束符而存在的陷阱（光盘位置：光盘\MR\Instance\4\18），下面代码要实现给一个指针变量动态分配一段空间，保存复制来的一个字符串。代码如下：

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void main()
```

```
{
    char *or_str="I am a supper man!";          //定义指针指向一个字符串
    char *cp_str=malloc(strlen(or_str));        //指针指向动态分配的空间
    strcpy(cp_str,or_str);                      //复制字符串
    printf("%s\n",cp_str);                      //输出
    free(cp_str);                               //释放内存
}
```

上面的代码运行程序后，显示要输出的字符串，但是同时弹出了一个异常提示信息。如图 4.30 所示。

2. 陷阱的分析

上面的代码使用 strlen(or_str)获取源字符串的长度，并使用这个长度动态分配了一个空间给指针变量，用于存储复制的字符串。表面看来没有什么错误，但是因为系统会自动给字符串添加一个结束标志“\0”，但是在分配空间的时候没有给这个结束符留出空间，在存储的时候这个结束符就超出了分配的内存边界，引起了异常。

注意：这种类型的错误很难发现。它可能不会引起一个异常，但是会给以后动态分配内存带来隐患，造成动态分配或者释放空间的失败。

3. 陷阱的解决方法

将分配的空间长度设置为字符串长度值加一，为字符串结束符留一个空间，这样存储的时候就不会出错了。具体代码如下：

```
char *cp_str=malloc(strlen(or_str)+1);          //分配字符串长度加一的空间
strcpy(cp_str,or_str);
```

程序的正确运行结果如图 4.31 所示。

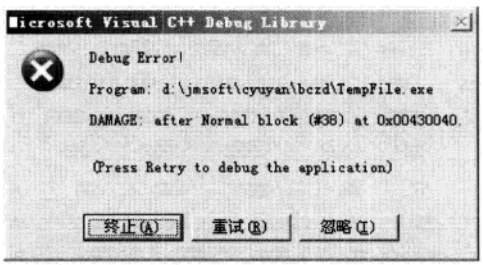


图 4.30　异常提示窗口

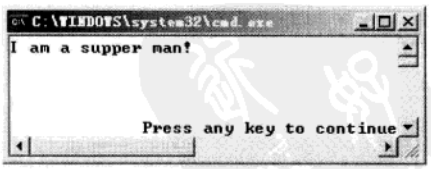


图 4.31　程序运行结果

陷阱 19　数组长度使用变量

1. 陷阱产生的场景

声明数组的时候需要指定数组的长度，因为在编译的时候需要确定声明的数组的长度进行分配

空间。有时候需要将数组的长度用一个标识符表示，如果使用了变量类型则会产生一个错误。

说明： 一般情况下，会将数组长度以一个整数常量来声明，因为通常情况下数组的长度会被多次用到，而且不可改变，所以声明为整数常量格式可以避免数组长度的前后不一致减少错误。

例 4.19 使用变量表示数组长度产生的错误（光盘位置：光盘\MR\Instance\4\19），代码如下：

```
#include "stdio.h"
void main()
{
    int i, max;;                          //声明变量
    int ts = 5;                           //定义变量
    int temp[ts] = {20,12,40,31,25};      //定义数组
    printf("数组 temp 的元素为：\n");      //输出
    for(i=0;i<ts;i++)                     //循环
    {
        printf("%d ",temp[i]);            //输出数组元素
    }
    printf("\n");                         //换行
    max=temp[0];                          //为变量赋值
    for(i=1;i<ts;i++)
    {
        if (max<temp[i])                  //条件判断
            max=temp[i];                  //得到较大值
    }
    printf("数组 temp 的最大值为：%d\n",max);
}
```

上面的代码运行程序后，显示了错误信息，如图 4.32 所示。

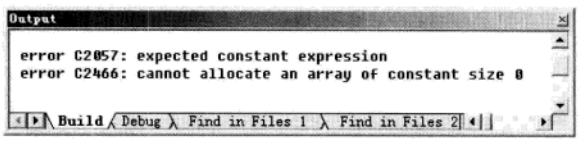


图 4.32 提示错误信息

2. 陷阱的分析

在进行数组定义时，数组的长度要使用常量或者符号常量，不可以使用变量。C 语言不允许对数组的长度做动态的定义。

说明： 如果有初始值，则数组长度可不指明，由括号内的元素数目自动决定。如果声明的数组长度大于给定的数组元素数目，则编译器将自动以零将不足的部分补满。

3. 陷阱的解决方法

将数组长度的变量定义为常量整型。具体代码如下：

```
const int ts = 5;
int temp[ts] = {20,12,40,31,25};
```

程序的正确运行效果如图 4.33 所示。

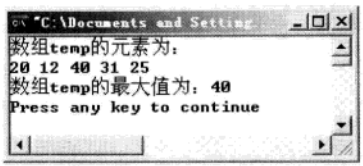


图 4.33　程序运行效果

陷阱 20　数组下标“越界”

1. 陷阱产生的场景

C 语言中的数组会根据数组长度自动分配存储空间，而对数组下标越界不进行检查。这样给程序开发带来灵活性，但同时也给程序留下陷阱。

说明：默认定义数组的长度的时候，用数组后面花括号中初始的数值大小来确定数组元素的个数。只有在进行带初始化的数组说明时才允许省略数组长度。

例 4.20　数组下标越界产生的陷阱（光盘位置：光盘\MR\Instance\4\20），代码如下：

```
#include "stdio.h"
void main()
{
    int i;                                    //声明变量
    int temp[5];                              //定义数组
    for(i=1;i<=5;i++)
    {
        temp[i]=i;                            //为数组元素赋值
    }
    printf("数组 temp 的元素为：\n");
    for(i=1;i<=5;i++)
    {
        printf("%d ",temp[i]);                //输出数组元素
    }
}
```

上面的程序代码运行后，不会产生错误信息。程序运行结果如图 4.34 所示，但是会为程序留下很大的隐患。

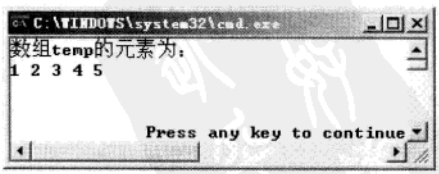


图 4.34　程序运行效果

2. 陷阱的分析

定义数组后，系统会根据数组长度为其分配相应的存储空间，数组下标规定是从 0 开始的，而上面例子中的 temp[5]，并不在数组的存储区内，系统可能会将其分配给其他程序使用。这里却给该位置的值进行了修改，可能会对其他程序甚至系统程序造成不可预料的后果，所以数组下标越界是后患无穷的。

说明：编程的时候常常会由于疏忽，将数组的长度写成了数组的最大下标值，从而引起数组下标越界的错误。

3. 陷阱的解决方法

切记数组元素下标是从 0 开始，数组的最大下标值为数组长度减一。本实例正确的代码如下：

```
#include "stdio.h"
void main()
{
    int i;                                  //声明变量
    int temp[5];                            //定义数组
    for(i=0;i<5;i++)                        //循环
    {
        temp[i]=i;                          //为数组元素赋值
    }
    printf("数组 temp 的元素为：\n");
    for(i=0;i<5;i++)
    {
        printf("%d ",temp[i]);              //输出数组元素
    }
}
```

陷阱 21 子函数未定义产生的陷阱

1. 陷阱产生的场景

自定义函数要在程序中定义之后才能使用。程序中使用了未定义的函数在进行编译的时候就会提示错误信息。

例 4.21 子函数未定义产生的陷阱（光盘位置：光盘\MR\Instance\4\21），本实例的关键代码如下：

```
#include <stdio.h>
void main()
{
    int a,b,c;
    scanf("%d","%d",&a,&b);                 //输入两个数值
    c=max(a,b);                             //调用求最大值函数
    printf("max is %d\n",c);                //输出
}
```

编译上面的代码，出现如图 4.35 所示的错误信息，提示 max 未定义。

```
warning C4013: 'max' undefined; assuming extern returning int
error LNK2001: unresolved external symbol _max
fatal error LNK1120: 1 unresolved externals
```

图 4.35 错误信息

2. 陷阱的分析

在编译的时候对一个未定义的函数进行调用系统会提示错误信息。这种错误与未定义变量的错

误类似，很容易被发现排除。

3. 陷阱的解决方法

在程序中定义求最大数的函数。代码如下：

```
int max(int x,int y)
{
    int z;                          //声明变量
    z=x>y ? x:y;                    //求出最大值
    return(z);                      //返回最大值
}
```

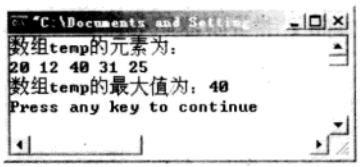


图 4.36　程序运行效果

程序的正确运行效果如图 4.36 所示。

陷阱 22　子函数在说明和定义时类型不一致

1. 陷阱产生的场景

在定义有参数的函数的时候，要求必须指定形参的类型。函数说明要与函数参数定义的类型相同，否则将产生错误。

例 4.22　子函数的说明与定义不符产生的错误（光盘位置：光盘\MR\Instance\4\22），本实例的关键代码如下：

```
#include <stdio.h>
void main()
{
    int max(float x,float y);           //函数说明
    int a,b,c;                          //声明变量
    scanf("%d,%d",&a,&b);               //输入两个整数
    c=max(a,b);                         //调用函数求最大值
    printf("max is %d",c);              //输出最大值
}
int max(int x,int y)                    //求最大值函数
{
    int z;                              //声明变量
    z=x>y ? x:y;                        //获取大的数值
    return(z);                          //返回最大值
}
```

运行上面代码，提示如图 4.37 所示警告信息。

```
warning C4244: 'function' : conversion from 'int ' to 'float ', possible loss of data
warning C4244: 'function' : conversion from 'int ' to 'float ', possible loss of data
 warning C4028: formal parameter 1 different from declaration
 warning C4028: formal parameter 2 different from declaration
```

图 4.37　错误信息

2. 陷阱的分析

自定义的函数需要将函数原型说明放在文件的开头部分，但是函数说明要与定义的类型一致，否则将会产生错误。

说明： 原型告诉编译器函数的两个重要特征：函数的返回类型和参数类型。如果没有提供原型，编译器会假设函数返回一个整型并可取多个任意类型的参数。所以自定义函数要将函数原型写在文件顶部，或者写进一个头文件中，在预定义的时候包含到程序中。

3. 陷阱的解决方法

将函数原型说明改为与函数定义的类型一致。代码如下：

```
int max(int x,int y);
```

程序正确的运行效果如图 4.38 所示。

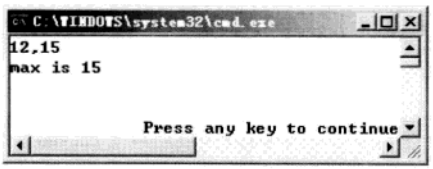


图 4.38　程序运行效果



第5章 36个开发人员意识中的常见谬误

读者朋友们在学习编程的过程中，很有可能对所学的知识产生不正确的理解，这种不正确的理解我们称之为谬误。

有了谬误要及时纠正，不然就有可能产生越来越多的问题。在本章，笔者将学习编程过程中容易犯的谬误进行了总结和概括，列出了开发人员意识中常见的谬误。这些谬误都是有一定深度的，希望读者朋友们在学习本章的内容之后不再会产生类似的意识偏差。

谬误01 Main大写错误

我们的都知道 C 语言是区分大小写的，如果某个字母被误写，则会造成不同的错误，如果在编写代码的过程中误将 main 函数中的 m 写成 M，将会产生如图 5.1 所示的错误提示。

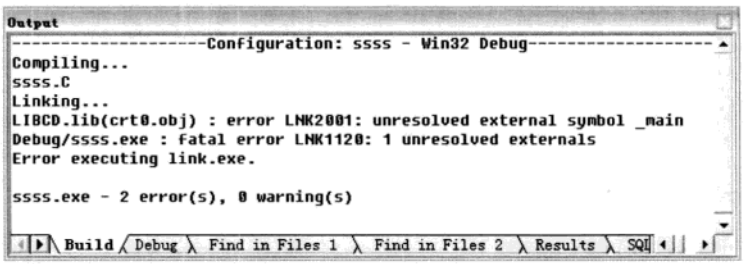


图 5.1 错误信息

解决这个错误的方法非常容易，只需将 M 改成小写即可。如果产生这个问题会不容易发现，因此读者在编写代码时一定要多加注意。

谬误02 “=”与“==”

符号=表示赋值运算，专门用来赋值并返回分配的值。符号==表示比较运算，专门用来比较并返回一个整型值（0 为 False，非 0 为 True）。如果在应该使用==的时候使用了=，由于存在返回值，C 编译器因此常常不能标记出这个错误。在实际的开发当中赋值运算比比较运算出现得频率高。在 C 语言中还可以使用重复赋值的形式，如，x=y=z。这种使用上的便利会导致一个潜在的问题，即当程序员本意是想执行比较操作时，却误写成赋值操作，这会使程序的意义发生改变，产生不必要

的错误，例如下面的代码：

```
if ( a = b )
break;
```

这段代码的意义是将 b 的值赋给 a，然后判断是否为 0，如果为 0，则跳出。而程序的愿意是判断 a 是否等于 b，如果相等则跳出。

又如，下面的代码：

```
int x = 5;
if ( x = 6 )
printf("x equals 6\n");
```

这段代码会输出 x equals 6！为什么？因为 if 内的赋值把 x 设置为 6 并把 6 返回给 if，既然 6 不是 0，就会被解释为 True。

有一种情况，放任何一个常量（或是任何右值表达式）在右边，编译器会发现这是一个错误。如下，if 里=的使用将是一个错误：

```
if ( 6 = x)
```

在许多高级语言中，使用“=”符号作为关系运算符“等于”。如在 BASIC 程序中可以写成：

```
if (a=5) then …
```

但 C 语言中，“=”是赋值运算符，“==”是关系运算符。如：

```
if (a==5) a=b;
```

前者是进行比较，a 是否和 5 相等，后者表示如果 a 和 5 相等，把 b 值赋给 a。由于习惯问题，初学者往往容易犯类似的错误。

谬误 03 scanf()错误

scanf()有两个常见的错误类型。

1. 忘记在参数上放一个&符号

scanf()必须有用于保存输入的变量地址。这意味着必须要常常使用取地址操作符来计算地址。

看下面的例子：

```
int x;
char * st = malloc(31);
scanf("%d", &x);            /* 传递一个地址给 scanf(), 符号&是必须的*/
scanf("%30s", st);          /* 这没有&, st 自己指向变量 */
```

上面代码的最后一行里没有&是正确的!

2. 使用了错误的操作数格式

C 编译器不能对所调用的 scanf()中参数使用的格式进行检查。大多数常见错误是对双精度数使用了%f 格式（必须使用%lf），以及混淆了字符与字符串%c、%s 格式的使用。

谬误 04　循环错误

在 C 语言里，一个循环语句为真，则循环下一条语句，例如，下面的代码：

```
int x = 5;
while( x > 0 );
x--;
```

可以看出上面的代码是一个无限循环。为什么这么说呢？因为 while 后的分号实际上定义了一个空语句，重复的是这个空语句（但它什么都不作）。移除分号才能让循环按我们的想法工作。

为避免循环错误，要认真的检查循环条件！

谬误 05　在定义数组时，将定义的“元素个数”误认为是可使用的最大下标值

在定义数组时，将定义的“元素个数”误认为是可使用的最大下标值。

```
main()
{
    static int a[10]={1,2,3,4,5,6,7,8,9,10};
    printf("%d",a[10]);
}
```

C 语言规定：定义时用 a[10]，表示 a 数组有 10 个元素。其下标值由 0 开始，所以数组元素 a[10]是不存在的。

谬误 06　混淆了字符与字符串常量

C 认为字符与字符串常量是不同的。字符常量被加上单引号而字符串常量被加上双引号。字符串常量相当一个指向实际字符串的指针。思考下面的代码。

代码：

```
char ch = 'A';                    /* 正确 */
char ch = "A";                    /* 错误 */
```

第二行把一个字符串常量的地址赋值给字符变量 ch。这应该会产生一个编译器错误。如果一个字符串指针被一个字符常量赋值也会出现同样的错误。

代码：

```
const char * st = "A";                    /* 正确 */
const char * st = 'A';                    /* 错误 */
```

谬误 07 使用==比较字符串

永远不要使用==运算符来比较字符串的值！字符串是字符数组。一个字符数组的名字相当于一个指向字符串的指针。为什么？思考下面的代码。

代码：

```
char st1[] = "abc";
char st2[] = "abc";
if ( st1 == st2 )
    printf("Yes");
else
    printf("No");
```

输出结果为 No。为什么？因为==比较的是 st1 和 st2 指针的值，而不是被它们所指向的数据。正确比较字符串的方法是使用 strcmp() 库函数（头文件应该包含 string.h）替换上面的 if 语句，代码如下：

```
if ( strcmp(st1,st2) == 0 )
    printf("Yes");
else
    printf("No");
```

输出结果为 Yes。

相同的原因，不要对字符串使用其他类似的逻辑运算符（<、>等）。

谬误 08 没有使用零字符结束字符串

C 假设一个字符串是以零字符（\0）结束的字符数组。这个零字符的 ASCII 值是 0 并可以使用 0 或'\0'来表示。这个值用来标记字符串内有效数据的结束。如果缺少这个值，很多 C 字符串函数将继续处理有效数据之后的数据，并且超过字符数组的边界，直到它自己在内存中找到一个零字节。

大多数 C 中的字符串函数创建的字符串将以零字符结束。不这样做的函数很少，例如有 strncpy()，因此要认真正确地使用。

谬误 09 不正确的使用 fgetc()等函数

fgetc()，getc()和 getchar() 函数全部返回一个整型值。例如 fgetc()的原型是：

```
int fgetc( FILE * );
```

有时这个整型值真的是一个简单的字符，但有一个非常重要的情况这个返回值不是一个字符！

而是“EOF”。学员常常有一个误解，认为在文件的结尾有一个特定的 EOF 字符。但实际上在文件的结尾并没有保存一个专门的字符。EOF 是函数返回的一个整型错误代码。下面是使用 fgetc()的错误习惯：

代码：

```
int count_line_size( FILE * fp )
{
    char ch;
    int cnt = 0;
    while( (ch = fgetc(fp)) != EOF && ch != '\n')
        cnt++;
    return cnt;
}
```

问题出现在 while 的循环条件中。举例来说，下面展示的是 C 语言在幕后重写的循环代码。

代码：

```
while( (int) ( ch = (char) fgetc(fp) ) != EOF && ch != '\n')
cnt++;
```

fgetc(fp)的返回值被转换为字符，结果保存到 ch 中。然后 ch 的值必须被转回到 int 与 EOF 进行比较。之后会怎样呢？转换一个 int 值到 char，然后再把一个 char 转换到 int，可能不会恢复为原始的 int 值。这意味着上述例子里如果 fgetc()返回的是 EOF 值，经过转换后，值可能被改变。而稍后与 EOF 进行的比较结果将会是 false。

解决方法是什么呢？下面代码把 ch 变量转换成 int。

代码：

```
int count_line_size( FILE * fp )
{
    int ch;
    int cnt = 0;
    while( (ch = fgetc(fp)) != EOF && ch != '\n')
        cnt++;
    return cnt;
}
```

不隐藏其中的转换，代码如下：

```
while( (ch = fgetc(fp)) != EOF && ch != ((int) '\n') )
cnt++;
```

所以经验是：所有 fgetc()，getc()和 getchar()的结果永远使用一个 int 变量保存。

谬误 10　a 和&a 的区别

输入变量时忘记加地址运算符“&”。

```
int a,b;
scanf("%d%d",a,b);
```

这是不合法的。scanf 函数的作用是：按照 a、b 在内存中的地址将 a、b 的值存进去。“&a”指 a 在内存中的地址。

谬误 11 gets()函数的使用

不要轻易使用这个函数！不能通过它安全地保存字符串中大量的字符，如果使用它来读取大量的字符，内存会被破坏。而事实上互联网上大量威胁安全的 Bugs 都是由这个原因造成的。可以使用 fgets()来代替它。但要记住，与 gets()不同，fgets()不会抛弃来自输入的终结符'\n'。

使用 scanf()有同样的危险。%s 格式接收的字符可能会超出目标串的长度。不过它可以指定所接收字符串的宽度来保证安全。例如使用%20s 格式使它读取的字符不会超过 20 个。

谬误 12 忽略了变量的类型，进行了不合法的运算

忽略了变量的类型，进行了不合法的运算。

```
main()
{
    float a,b;
    printf("%d",a%b);
}
```

%是求余运算，得到 a/b 的整余数。整型变量 a 和 b 可以进行求余运算，而实型变量则不允许进行“求余”运算。

谬误 13 忘记加分号

分号是 C 语句中不可缺少的一部分，语句末尾必须有分号。

```
a=1
b=2
```

编译时，编译程序在“a=1”后面没发现分号，就把下一行“b=2”也作为上一行语句的一部分，这就会出现语法错误。程序报错时，有时在被指出有错的一行中未发现错误，就需要看上一行是否漏掉了分号。

```
{
    z=x+y;
    t=z/100;
    printf("%f",t);
}
```

对于复合语句来说，最后一个语句中最后的分号不能忽略不写（这是和 PASCAL 不同的）。

谬误 14　路径名错误

在 MS-DOS 中，斜杠\表示一个目录名，而在 Turbo C 中斜杠是个某个字符串的一个转义字符。这样，在用 Turbo C 字符串给出一个路径名时应考虑"\"的转义的作用。例如，有这样一条语句：

```
file=fopen("c:\new\tbc.dat", "rb");
```

目的是打开 C 盘中 new 目录中的 tbc.dat 文件。但无法按预期运行。这里"\"后面紧接的分别是"n"及"t"，"\n"及"\t"将被分别编译为换行及 tab 字符，DOS 将认为它是不正确的文件名而拒绝接受，因为文件名中不能有换行或 tab 字符。正确的写法应为：

```
file=fopen("c:\\new\\tbc.dat", "rb");
```

谬误 15　#include <stdio.h>和#include "stdio.h"的区别

在编写 C 语言的程序时，需要使用到头文件，在引用时，会有如下两种形式：

```
#include "stdio.h"
#include <stdio.h>
```

这两种形式都是正确的，在使用时，有着细小的差别。

- #include "stdio.h"

这种形式的引用当调用某函数时，系统会先在自己编写的文件中进行查找，如果找不到，就到库文件中查找。

- #include <stdio.h>

这种形式的引用，当调用某个函数时，系统就会直接在库文件中进行查找。

所以，如果是调用自己写的函数的话，就用#include "stdio.h"的形式，如果是调用标准的库函数就用#include <stdio.h>的形式，这样可以提高执行的效率。

谬误 16　函数调用的错误

在 C 语言中调用函数时，即使不带参数，也需要包含参数列表。例如，假设 fac()是一个函数，那么下面的语句是一个函数调用的语句：

```
fac();
```

而下面的代码却是一个什么也不做的语句，如果说得更准确些，下面的语句将计算 fac()函数的地址，并不是调用 fac()这个函数。

```
fac;
```

谬误 17 空指针并非空字符串

一般情况下，C 语言中将一个整数转换为一个指针，得到的结果取决于具体的编译器。对于常数 0，编译器将保证由 0 转换来的指针不等于任何有效的指针。在使用时常数 0 经常用符号 NULL 来代替。

```
#define NULL 0
```

在使用时，无论使用常数 0，还是使用符号 NULL 在效果上都是一样的。但是，当常数 0 被转换为指针使用时，这个指针绝对不能被解除引用。换句话说，当将 0 赋值给一个指针变量时，绝对不能使用该指针变量所指向的内存中的内容。

下面的写法是正确的写法：

```
if (p==(char *)0)
```

而下面的写法，则是不正确的。

```
if (strcmp(p,(char *)0)==0)
```

原因是在于库函数 strcmp 的实现中会包括查看它的指针参数所指向的内存操作。

谬误 18 枚举类型容易出错的几个问题

（1）在 C 语言中，将枚举元素按照常量进行处理，因此也称为枚举常量。既然是常量就不能对它们进行赋值。例如，下面的代码，如果出现在枚举类型定义之后就是错误的。

```
red=2;black=0;
```

（2）枚举元素是有值的常量，在没有赋值的情况下，会按照定义时的顺序按照 0、1、2、3、……进行赋值。

（3）枚举值可以用来做比较判断。例如：

```
if (workday == mon)    …
if (colorball > black)    …
```

（4）在给枚举变量赋值时，不能使用整数。例如，下面的代码是不正确的。

```
workday = 2;
```

因为它们属于不同的类型，如果进行强制类型装换后就可以进行赋值了，代码如下：

```
workday = (enum weekday)2;
```

这个代码相当于将序号为 2 的枚举元素赋给 workday 变量，跟下面的赋值效果是一样的。

```
workday = tue;
```

谬误 19　&、|、~与&&、||、! 的谬误

按位运算符（&、|、～）与逻辑运算符（&&、||、!）属于两类不同的运算符，在某些特殊的情况下可以互换，而且在交换以后程序还能“正常”工作。但是，事实上这仅是一种特殊的情况而已，或者说纯属巧合。

进行按位运算（&、|、～）的操作数被默认为是一个二进制的位序列，分别对其中的每个位进行操作。

而逻辑运算符（&&、||、!）将操作数当成非真即假、非假即真。通常都是将 0 当成假，非 0 即为真。

谬误 20　system()和 clrscr()的区别

它 clrscr()函数是 Turbo C 中所特有的，用于用户清屏的函数。尽量使用 system(“cls”)来取代它，system(“cls”)函数的通用性和兼容性更好些。

System()函数在# include <stdio.h>头文件中。

谬误 21　使用 getchar()函数的误区

getchar()函数作用是从终端输入一个字符。有一点要注意，getchar 每次获取一个字符。

getchar 函数的一般形式为：

```
getchar();
```

函数的返回值就是从输入设备得到的字符。

下面来看一个例子：

```
#include <stdio.h>
main()
{
    char c;
    while ((c=getchar())!=EOF)
        putchar(c);
}
```

getchar()函数用于从终端输入一个字符，当没有输入时，则返回 EOF，这个 EOF 是在头文件 stdio.h 中被定义的，不同于任何一个字符。

这段程序似乎是将标准输入复制到标准输出中，实际上并不是这样的。程序中变量 c 被声明为 char 类型，而不是 int 类型。这意味着 c 无法容下所有可能的字符，特别是 EOF。

因此，最终的结果可能有两种可能。一种，可能某些合法的输入字符在被“截断”后使得变量

c 的取值与 EOF 相同；另一种，可能变量 c 根本无法取到 EOF 这个值。对于前一种情况，程序将在文件复制的中途终止；对于后一种情况，程序将陷入一个死循环。

实际上，还有第三种情况：程序表面上似乎能够正常工作。不过完全是因为巧合。尽管函数 getchar()的返回结果在赋给 char 类型的变量 c 时会发生“截断”现象，尽管 while 语句中比较运算的操作数不是函数 getchar()的返回值，而是被“截断”的变量 c 的值，然而许多编译器对上述表达式的实现并不准确。这些编译器确实对函数 getchar()的返回值做了“截断”操作，并把低端字节部分赋给函数的返回值与 EOF，编译器如果采取的是这种做法，上面的程序就能正确运行了。

谬误 22 使用 errno 检测错误

很多库函数在使用时，当执行失败时会通过一个名称为 errno 的外部变量通知程序函数调用失败。

看下面的代码：

```
/*调用函数*/
if(errno)
    /*处理错误*/
```

上面的代码是利用 errno 的特性设计的，似乎是正确的，但事实上并不是。出错的原因在于，在库函数调用没有失败的情况下，并没有强制要求库函数一定要设置 errno 为 0，这样 errno 的值就可能是前一个执行失败的库函数的设置值。

看下面这个更正了的代码：

```
errno=0;
/*调用函数*/
if(errno)
    /*处理错误*/
```

上面的代码似乎可以正常工作，但是实际上还是有错误的，库函数在调用成功时，既没有强制要求对 errno 清零，但是同时也没有禁止设置 errno。既然库函数已经设置成功，为什么还有可能设置 errno 呢？要想理解这一点，我们不妨设想一下库函数 fopen 在调用时可能会发生什么情况。当 fopen 函数被要求新建一个文件提供程序输出使用，如果已经存在一个同名文件，fopen 函数会先删除此文件，然后再创建一个新文件。这样，fopen 可能需要调用其他的库函数，用于检测同名的文件是否存在。假设用于检测文件的库函数在文件不存在时会设置 errno。那么 fopen()函数每次新建一个实现并不存在的文件时，即使没有任何程序错误发生，也会设置 errno。

因此，在调用库函数时，应该先检测作为错误提示的返回值，确定程序执行已经失败。然后，再检查 errno，来搞清楚出错的原因。

```
/*调用函数*/
if(返回的错误值)
    /*检查 errno*/
```

谬误 23　宏定义中不可忽视的空格

在使用函数的时候，如果函数不带参数，在调用时只需在函数名后面加一对括号即可。而一个宏如果不带参数，则只需要使用宏名即可，括号则无关紧要。只要宏已经定义过了，就不会带来什么问题：预处理器从宏定义中就可以知道宏调用时是否需要参数。

与宏调用相比，宏定义则有更多需要注意的地方。例如，下面的宏定义中 f 是否带了一个参数呢？

```
#define f (x) ((x)-1)
```

答案有以下两种可能。

一种是 f(x)代表 ((x)-1)；另一种是 f(x)代表(x)((x)-1)。

而在上述的定义中，编译器默认第二种形式是正确的形式。原因是因为在 f 的后面多了一个空格。所以如果想定义为第一种的形式，需要将空格去掉，写成下面的形式：

```
#define f(x) ((x)-1)
```

这一原则在宏定义的时候要特别注意，但是在宏调用的时候却不受此影响。例如，f(3)和 f　(3)在求值以后都等于 2。

谬误 24　*p++是自增 p 还是 p 做指向的变量

后缀++和--操作符的优先级要高于前缀的一目运算符的，因此*p++和*（p++）是等价的，自增 p 并返回 p 自增之前所指向的值。要想自增 p 所指向的值，需要使用（*p）++，如果副作用的顺序无关紧要，则也可以使用++*p 的形式。

谬误 25　char a[]和 char *a 是一样的

char a[]和 char *a 并不是一样的。数据跟指针不是一回事。数据定义了 char a[5]，即请求预留 5 个字符的位置，并用名称“a”表示。也就是说，有一个称为“a”的位置，可以放入 6 个字符。而指针声明 char *p，请求一个位置放置一个指针，用名称“p”表示。这个指针几乎可以指向任何一个位置：任何字符和任何连续的字符串，或者哪里也不指向。

通过图 4.4 来使大家加深理解。

声明形式为：

```
char a[]="mrsoft";
char *P="mrsoft";
```

初始化的效果如图 5.2 所示。

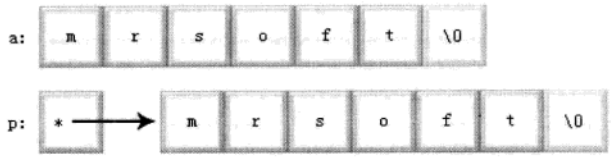


图 5.2　初始化效果图

谬误 26　指针变量与零值的比较

指针变量的零值是“空”，写为 NULL。尽管 NULL 的值与 0 相同，但是两者意义并不相同。在使用时应该将指针变量用“==”或者“！=”与 NULL 相比较。

例如，假设指针变量 p，它与 0 相比的标准的 if 语句如下：

```
if (p==NULL)                    /*p 与 NULL 显示比较，强调 p 是指针变量*/
if (p!=NULL)
```

下面的形式，会让大家误解当前的 p 值是一个整型变量

```
if (p==0)
if (p!=0)
```

下面的代码也不是正确的写法，这种写法会让人误解 p 是一个布尔变量

```
if (p)
if (!p)
```

谬误 27　const 和#define 是一样的

C 语言中 const 和#define 都能用来定义常量，但是 const 和#define 却并不是一样的。const 比#define 有更多的优点。

（1）const 定义的常量是有数据类型的，而利用#define 定义的宏常量没有数据类型。编译器可以对前者进行类型安全检查，而后者进行字符替换，没有类型安全检查，并且在字符替换时，有可能会产生一些意外的错误。

（2）在一些集成化的调试工具中可以对 const 定义的常量进行调试，但是不能对宏常量进行调试。

在使用时，应该尽量使用 const 声明常量而不使用#define 声明常量，即使用 const 声明常量完全可以取代宏常量。

谬误 28　浮点变量与零值的比较

在使用浮点变量的时候，不能将浮点变量用“==”或“！=”与任何数字比较。因为无论是 float

还是 double 类型的变量都是具有精度限制的，所以一定要避免将浮点变量用“==”和“！=”与数字进行比较，应该设法转化为“>=”或“<=”形式。

例如，假设一个浮点变量的名字为 a，则下面利用“==”的代码是不正确的。

```
if(x==0.**)
```

应该将上面的代码转换为“>=”或“<=”形式。

```
if((x>=-0.**) && (x<=0.**))
```

其中，0.**就是允许的误差或者精度。

谬误 29　宏定义中的常见谬误

有时在使用一些定义函数式的宏的时候会出现一些问题，但有的时候又没有问题，这就在定义时注意一些问题。

- 宏扩展必须使用括号，来保护表达式中低优先级的操作符，以便确保调用时达到想要的效果。

例如：

```
#define ADD(n) n+n
```

调用：

```
5*ADD(n)
```

则会被扩展为：

```
5*n+n
```

而本意是希望得到：

```
5*(n+n)
```

解决的办法：宏扩展时加上括号就能避免这种错误发生。

- 在宏定义内部，所有参数的出现都必须用括号括起来，以便保护实参中任何低优先级的操作符不受宏扩展其他部分的影响。

例如：

```
#define MUL(n) n*n
```

调用：

```
MUL(n+1)
```

则会被扩展为：

```
n+1*n+1
```

而本意是希望得到：

```
(n+1)*(n+1)
```

- 如果一个参数在扩展中出现多次且实参是带其他作用的表达式，则宏将不能正确的运行。

例如：

```
#define MUL(n) n*n
```

调用：

```
MUL(i++)
```

则会被扩展为：

```
i++*i++
```

在同一个表达式中使用导致同一对象修改两次或修改以后又被检查的自增或自减。在赋值操作符时，则认为是未定义的。

谬误 30 a[i]=i++;代码正确吗

语句 a[i]=i++；是否能正常执行呢？答案很显然是不能。因为子表达式 i++有一个副作用，就是它会改变 i 的值，由于 i 在同一个表达式的其他地方被引用，这将导致一个无法判断的结果，无从判断 a[i]中的 i 是旧值还是真值。在使用中应该避免此类现象的出现。

谬误 31 数组和指针等价吗

在 C 语言中数组和指针等价吗？在学习的过程中经常会听到这样的说法。这里面说的数组和指针“等价”并不表示他们是相同的，他们不能互换，其含义是数组和指针的算法定义，可以用指针方便的访问数组。

经常说数组和指针是等价的原因是：一个 p 的数组类型的左值如果出现在表达式中会蜕变为一个指向数组第一个成员的指针，结果指针的类型是 p 的指针。换句话说，一定数组出现在表达式中，编译器会隐式的生成给一个指向数组第一个成员的地址，就像程序员写出&a[0]一样。

谬误 32 指针和数组的区别

数组自动分配空间，但是不能重分配或者改变大小。指针必须明确赋值以指向分配的空间，但是可以随意重新赋值，同时除了表示一个内存块的基址以外，还有许多其他的用途。

由于数组和指针的所谓等价性，数组和指针经常看起来可以互换，而事实上指向 malloc 分配的内存块的指针通常被看作一个真正的数组。

谬误 33　指针比数组执行快

指针真的比数组执行的快吗？这个问题跟开发者所使用的处理器和编译器有关。通常情况下，差别是很小的，小到要经过千万次的迭代才能看出不同来。

一般的机器，通常遍历大的数组时，用指针比用数组要快，但是对于某些处理器则是相反的。

谬误 34　++i 比 i=i+1 执行快

在解释这个问题之前，首先要了解我们是在跟编译器打交道，因此对于++i 或者 i=i+1 这样的代码，任何编译器都会产生一样的代码。所以程序员使用任何一种代码的写法都只是跟编程的风格有关，而跟程序的执行效率无关。

3

第 3 篇
技术细节
——修炼编程内功心法

第 6 章　26 个 C 语言基础知识技术细节
第 7 章　19 个 C 语言重点知识技术细节
第 8 章　18 个 C 语言高级技术细节





第6章 26个C语言基础知识技术细节

C语言基础是衡量程序员基本素质的硬性指标。本节列出C语言开发过程中典型的技术细节。每个技术细节都具有一定的代表性和深度，相信读者朋友们学习完本章的内容后会对C语言有一个本质的认识，同时自身技术水平得到升华。

细节01 标识符的使用

所谓标识符是指常量、变量、语句标号以及用户自定义函数的名称。

编程准则：标识符应该遵循的原则

作为标识符必须满足以下规则：

1. 所有标识符必须由字母或下划线开头
2. 标识符的其他部分可以用字母、下划线或数字组成
3. 大小写字母表示不同意义，即代表不同的标识符
4. 标识符不能是关键字

例如：ac.12、37_ab、￥12、ab#

以上标识符均是不合法的。

又如下面列举的标识符均是合法的：

_abc、xy12、_12a、day、sum。

规则中第一条规定了标识符只能是字母或下划线，其他均不可，所以上面的37_ab是不合法的标识符。根据第二条的规定可判断出￥12和ab#均是不合法的标识符，因为“￥”和“#”不是字母、下划线和数字中的任意一个。规则中的第三条表明C语言中的字母是区分大小写的，因此sum、Sum和SUM是三个不同的标识符。

ANSI标准规定，标识符可以为任意长度，但外部名必须至少能由前8个字符唯一地区分。这

是因为某些编译程序（如 IBM PC 的 MS C）仅能识别前 8 个字符。

标识符不能和 C 语言的关键字相同，也不能和用户已编写的函数或 C 语言库函数同名。

细节 02　限定词 const

限定词 const 在 C 语言中的主要作用是定义只读变量。只读变量的值在定义后不能再改变。使用 const 定义的形式如下：

```
const 数据类型名 变量名=初始值;
```

也可以把关键字 const 放在数据类型名和变量名中间，定义形式如下：

```
数据类型名 const 变量名=初始值;
```

该变量被定义为只读变量，在程序中不能被改变。定以 const 变量时，必须要对该变量初始化；否则编译器会报错。

例如：

```
const int NUMBE=12;
const char SIGN='R';
```

上面定义的 NUMBER 和 SIGN 都是 const 变量，不能被修改。

说明：习惯上将 const 变量名全部大写。

使用 const 变量可以避免只读变量被意外修改，在程序中如果修改了 const 定义的只读变量，则编译器就会报错。使用 const 变量的另一个好处就是大多数编译器不会为 const 变量分配存储空间，而是将它们保存在符号表中，这样就不需要读写内存的操作。

const 限定的对象通常是运行时不能被赋值的对象，因此用 const 限定的对象的值并不是一个真正的常量，不能用作数组维度等。

例如：

```
const int n=10;
int a[n];
```

根据上面的讲解可知道，上面这两行代码有问题，不能使用 const 定义的常量来做数组的维度。

细节 03　限定词 volatile

限定词 volatile 修饰的变量是随时可能变化的，volatile 的使用语法和 const 一样，其定义形式如下：

```
volatile 数据类型名 变量名;
```

例如

```
volatile int num=5;
```

或者将 volatile 放在数据类型名和变量名的中间，定义形式如下：

```
数据类型名 volatile 变量名
```

num 被定义为 int 型的 volatile 变量。

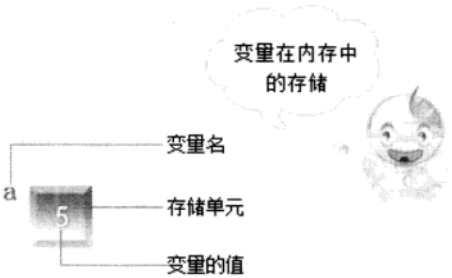
volatile 经常用在多线程环境下，因为在编写多线程的程序时，同一个变量可能会被多个线程修改，通过定义 volatile 变量，程序在读取该变量前会同步各个线程，以便取得该变量的最新修改值。

细节 04 变量的存储

在程序的执行过程中，其值是可以改变的量称为变量。一个变量有一个指定的名字，称为“变量名”。变量在内存中占用一定的存储空间，这个存储空间被称为“存储单元”，在存储单元中存放的是变量的值。变量名和变量的值是两个不同的概念。变量名是一个存储空间，一个地址的标识符号。在程序编译连接时，系统给每个变量分配一个内存空间，在这个内存空间中存储的是变量的值。程序执行时，如果要读取变量的值，先根据变量名找到对应的存储单元，再从这个存储单元中读取变量的值。

我们可以从 3 个方面来理解变量：变量名，存储单元和变量的值。

- 变量名：标记内存空间的符号地址，一个变量名对应一个存储单元和一个变量的值。
- 存储单元：变量名所对应的存储空间。
- 变量的值：存储单元所存放的数据。



细节 05 字符数据类型

一般每一个字符型（char）声明的变量所占据的存储空间是 8 位，因为 2^8=256。所以每个字符

可以代表 256 个不同的值，C 语言系统中，这 256 个不同的值是根据 ASCII 码的值排列的，而这些码的值包含了小写字母、大写字母、数字、标点符号以及其他一些特殊符号。

例如，声明一个字符变量 name_char，它声明形式如下：

```
char name_char;
```

又如，声明一个字符变量 name_char，并将其赋值为'mrsoft'，声明形式如下：

```
char name_char ='mrsoft';
```

另外，在 C 语言里面还有一些无法打印的字符，例如，'\0'，虽然在单引号中有\和 0，但是他们合起来算是一个字符，我们又称这些字符为换码（escape）字符，如表 6.1 所示。

表 6.1　　常见换码字符

整 数 值	字符表示方式	字 元 名 称
0	\0	空格（null space）
7	\a	响铃（bell ring）
8	\b	退格（backspace）
9	\t	标识（tab）
10	\n	新行（newline）
12	\f	换页（form feed）
13	\r	回转（carriage return）
34	\"	双引号（double quote）
39	\'	单引号（single quote）
92	\\	反斜杠（back slash）

注意：并不是每种计算机对于用 char 声明的变量都是 8 位的。

细节 06　说明变量和定义变量

说明一个变量实质上是在向编译程序描述变量的类型，但并不为变量分配存储空间。

定义一个变量则是在说明变量的同时还要为变量分配存储空间。在定义一个变量的同时还可以对变量进行初始化。

说明一个变量相当于告诉编译程序“在程序的某个位置将用到一个变量，这里给出了它的名称和类型”。一个变量可以被说明多次，而定义一个变量则相当于告诉编译程序“具有这个名称和这种类型的变量就在这里”。注意，一个变量只能被定义一次，而不像说明可被说明多次。

注意：不应该在头文件中定义变量，因为一个头文件可能会被一个程序的许多源文件所包含。

细节 07 typedef 应用的技术要点

（1）使用 typedef 关键字定义数据类型，只是对享有的数据类型进行重命名，或者说添加一个别名，并不是说创建了一个新的数据类型。例如，前面介绍的声明整型类型 INTEGER，只是给 int 类型起了一个别名，用 INTEGER 来表示，并没有创建一个新的数据类型。

（2）使用 typedef 关键字可以用来定义各种数据类型，但是不能用来声明变量，在使用 typedef 关键字声明数组类型、字符串类型时，是比较方便的。

例如，声明一个大小为 10 的整型数组 a，程序代码如下：

```
int a[10], b[10],c[10], d[10], e[10];
```

上面的代码声明了 5 个大小为 10 的整型数组，由于这几个数组都是相同的数据类型，大小也相同，可以将这些数组的类型声明为一个类型，然后用这个类型去定义数组。声明数组类型的代码如下：

```
typedef int arr[10];
```

然后，用 arr 去声明 a、b、c、d、e 这几个数组，形式如下：

```
arr a,b,c,d,e;
```

这里面 arr 为数组类型，包含 10 个数组元素，所以 a，b，c，d，e 都是具有 10 个数组元素的整型数组。

从这段代码中可以看出，使用 typedef 关键字将数组类型和数组变量分离开了，利用这个数组类型可以声明其他数组变量。同样的，这种方法也适用于定义指针类型和字符串类型。

（3）当在不同的程序文件中使用同一类型的数据时，例如，像前面定义的数组。经常使用 typedef 关键字来声明数据类型，并且将这些声明单独存放在一个文件中，当需要使用的时候，使用#include 命令把他们引入进来就可以了。

（4）使用 typedef 关键字声明数据类型有利于程序的移植。例如，有些计算机中 int 类型占用 2 字节，而在另外的计算机中 int 数据类型占用 4 字节的存储空间，如果将 1 个程序从 4 字节的计算机中移植到 2 字节的计算机中，在 4 字节的计算机中 int 类型为 4 个字节，在 2 字节的计算机中可以使用 long 型，因此需要将原程序中的 int 类型改为 long 型。在程序中可能有多处使用到了 int 定义变量，如果一一修改比较麻烦，可以在编写代码时，将 int 类型用 INTEGER 来代替，如下面的代码：

```
typedef int INTEGER;
```

在程序移植时，只需将 typedef 定义的类型修改为下面的形式即可。

```
typedef long INTEGER;
```

细节 08　使用 static 关键字声明静态变量

编写程序，有时候需要在调用函数中的某个局部变量以后，这个变量的值不消失，并且保持原值不变，也就是该变量所占用的存储空间不被释放，在下次调用该函数的时候，变量中的值是上次调用该函数结束时变量的值。这时使用的变量类型是静态变量，他使用 static 关键进行声明，静态变量属于静态存储方式。

定义变量时，使用 static 关键字就可以将变量定义为静态变量，形式如下：

```
static 类型说明符 变量 1，变量 2 …
```

用 static 关键字声明的外部变量会得到静态全局变量，或者称为静态外部变量。当用 static 关键字定义内部变量时，会得到静态局部变量，或者称为静态内部变量。

例 6.1　使用 static 关键字声明静态变量（光盘位置：光盘\MR\Instance\6\01）。下面通过一个例子来理解静态变量。

```
#include <stdio.h>
test()
{
    auto int a=0;                          /*定义自动存储类型变量 a*/
    static b=3;                            /*定义静态存储类型变量 b*/
    a++;                                   /*变量 a 自加 1*/
    b++;                                   /*变量 b 自加 1*/
    printf("%d\n",a);                      /*输出变量 a 的值*/
    printf("%d\n",b);                      /*输出变量 b 的值*/
}

main()
{
    int i;                                 /*定义整型变量 i，循环计数*/
    for(i=0;i<3;i++)                       /*循环 3 次*/
        test();                            /*调用自定义函数*/
}
```

程序的运行结果如图 6.1 所示。

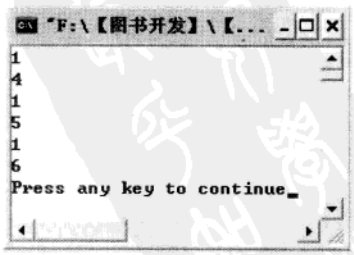


图 6.1　程序运行结果

在上面的例子中，一共调用 test 函数 3 次。在第一次调用 test 函数的时候，变量 a 的值是 0，变量 b 的值是 3。调用结束以后，变量 a 的值为 1，变量 b 的值为 4。第二次调用时，变量 a 的值是 0，变量 b 的值是 4，因为 a 是自动变量，函数调用结束以后存储空间的值被释放，因此在第二次调用的时候，使用的是函数的初值，变量 b 被定义为静态类型的变量。在第一次调用函数以后，变量的值保持不变，在第二次调用的时候，变量 b 的值就是上次调用结束时的值，因此 b 的值为 4。

在第二次调用结束以后，变量 a 的值为 1，变量 b 的值为 5。先后调用 3 次 test 函数，变量 a 和变量 b 的值如图 6.2 所示。

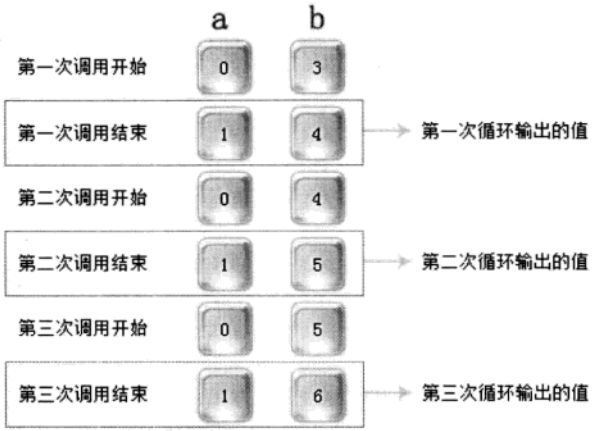


图 6.2　变量 a 和变量 b 的变化过程

细节 09　如何区分“,”是运算符还是分隔符

“,”作为分隔符主要用于以下情况。

（1）变量声明时使用逗号分隔多个变量名。

（2）函数有多个参数时，用逗号分割参数。

逗号运算符是 C 语言提供的一种特殊的运算符，用它可以将两个表达式连接起来。例如，“1+2，4+5”，称为逗号表达式，也称为“顺序求值运算符”，表达式一般形式为“表达式 1，表达式 2”，执行顺序是先计算表达式 1 的值，再计算表达式 2 的值。表达式也可以扩展为“表达式 1，表达式 2，表达式 3，…表达式 n”。

例如：

```
int min(a,b,c);
int a,b,c;
```

在这两句代码中逗号都是起到分隔符的作用，在第一句代码中逗号用来分割多个函数参数，第二句代码中逗号分隔多个变量名。

又如：

```
int x,y,z;
x=1, y=2, z=3;
```

这两句代码中，第一句中逗号起到的是分隔符的作用。第二句代码中的逗号是起到运算符的作

用，相当于下面的代码：

```
x=1; y=2; z=3;
```

一般在只允许出现一个表达式的地方却出现了多个表达式，并且是用逗号相连接的，这种表达式就是逗号表达式。此时逗号就是运算符。

下面这个例子能加深读者对逗号表达式和逗号分隔符的理解。

```
main()
{
int a,b,c,d;
c =(a=3,2*a);
d=a=b=3,2*a;
printf("%d,%d,%d,%d\n",a,b,c,d);
}
```

程序的执行效果如图 6.3 所示。

在上面的代码中，代码 c =(a=3,2*a);是将一个逗号表达式的值赋给变量 c，第一个表达式 a=3，第二个表达式 2*a，计算得到 6，因此变量 c 的值是 6.。

代码 d=a=b=3,2*a;整体是一个逗号表达式。这个逗号表达式的值同样是 2*a 的值，结果是 6，但是变量 d 的值是在第一个表达式中计算得到的，得到的值为 3。

如果将代码改为下面的形式：

```
d=(a=b=3,2*a);
```

这样变量 d 得到的值就是逗号表达式的值，即表达式 2*a 的值，运行效果如图 6.4 所示。

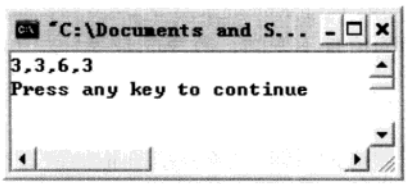


图 6.3　程序的执行效果

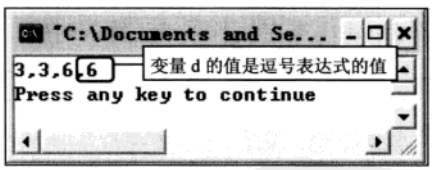


图 6.4　变量 d 的值变为 6

细节 10　基本的算术运算

和任何一种编程语言一样，C 语言也具有基本的算术运算，这些基本的算术运算有加、减、乘、除和取余运算。执行这些运算需要相对应的运算符，这些运算符在执行运算的时候都需要两个变量进行运算，因此称这些运算符为“二目运算符”。下面就对这些运算符和相应的运算进行介绍。

1. 加法

在 C 语言中执行加法的运算符为“+”，他的主要功能是将两个数的值相加。

例如，下面的语句就是用来执行两个数相加的：

```
sum=a+b;
```

在上述代码中 sum 是两个数相加得到的和，而变量 a 和 b 即为两个加数，+为运算符。如果在执行前假设，a=2，b=3，sum=0。当程序执行完这条语句以后得到的结果是，a=2，b=3，sum=5。从中我们就可以看出来在执行加法运算的时候，两个加数的值是不变的，改变的是等号左边的值。

2. 减法

在 C 语言中用于执行减法的运算符是“-”，他的主要功能是将第一个运算变量的值减去第二个运算变量的值。

例如，下面的语句就是用来执行两个数相减的。

```
minus=a-b;
```

假设在执行前变量值：a=3，b=2，minus=0。当程序执行完这一条语句以后，得到的结果是 a=3，b=2，minus=1。我们得到的结果同样是在执行运算前运算变量是不变的，而结果值是在变化的。

3. 乘法

在 C 语言中执行乘法的运算符为“*”，他的主要功能是将两个数的值相乘。

例如，下面的语句就是用来执行两个数相乘的。

```
multi=a * b;
```

在上述代码中 multi 是两个数相加得到的和，而变量 a 和 b 即为两个乘数，*为运算符。如果在执行前假设，a=2，b=3，sum=0。当程序执行完这条语句以后得到的结果是 a=2，b=3，sum=6。从中我们就可以看出来在执行乘法运算的时候，两个乘数的值是不变的，改变的是等号左边的值。

4. 除法

在 C 语言中用于执行除法的运算符是“/”，他的主要功能是将第一个运算变量的值除以第二个运算变量的值。

例如，下面的语句就是用来执行两个数相除的。

```
div=a / b;
```

假设在执行前，变量值 a=6，b=2，div=0。当程序执行完这一条语句以后，得到的结果是 a=6，b=2，div =3。我们得到的结果同样是在执行运算前运算变量是不变的，而结果值是在变化的。

5. 取余

在 C 语言中用于执行取余的运算符是“%”，他的主要功能是将第一个运算变量的值除以第二个运算变量，然后得到余数。

例如，下面的语句就是用来执行两个数相除的。

```
surplus=a % b;
```

假设在执行前变量值：a=7，b=3，surplus =0。当程序执行完这一条语句以后，得到的结果是 a=7，b=3，surplus =1。我们得到的结果同样是在执行运算前运算变量是不变的，而结果值是在变化的。

除了以上五种基本运算以外，C 语言还提供取负运算（-）。这个运算符号的表达方式和减号一样，但是意义却不同，前面已经说过减法运算符需要有两个运算量进行搭配，而取负运算符号只需要一个运算量就可以了。根据这个特性，人们将其称为一元运算符或者单目运算符。

例如，下面的一句代码：

```
sum = -a + b;
```

假设在执行前，a=3，b=2，sum=0。当程序执行完这条语句以后得到的结果 a=3，b=2，sum=-1。和前面的范例一样，运算量本身在执行前后没有发生变化。

也许有些读者会问，为什么要先算-a，而不是先计算 a+b，然后再计算取负运算？其实原因很简单，不同的运算符有不同的运算顺序，即优先级不同，图 6.5 就给出了这 6 中运算符号的优先顺序。

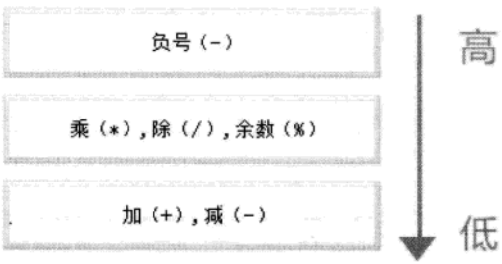


图 6.5　运算符的优先级

通过上图介绍，相信读者一定能够了解前面的范例为什么会先计算-a，而不是 a+b。

细节 11　自增、自减操作符的使用

在 C 语言中有两个很有用的运算符，在其他高级语言中通常没有。这两个运算符就是增 1 和减 1 运算符“++”和“--”，运算符“++”使操作数加 1，而“--”则使操作数减 1。自增 1，自减 1 运算符均为单目运算，都具有右结合性。有以下几种形式。

++i：i 自增 1 后再使用 i。

--i：i 自减 1 后再使用 i。

i++：使用 i 后 i 的值再自增 1。

i--：使用 i 后 i 的值再自减 1。

上述四种形式在使用时经常会出现错误，通过一个例子来了解一下自增、自减运算符使用方法。

例 6.2 自加自减运算符应用（光盘位置：光盘\MR\Instance\6\02）。

```
#include<stdio.h>
main()
{
    int i=5,x1,x2,x3,x4;                    /*定义变量为基本整型，并为部分变量赋初值*/
    x1=i++;                                 /*先将 i 值赋给 x1，然后加 1*/
    x2=++i;                                 /*将 i 值加 1 后的结果赋给 x2*/
    x3=i--;                                 /*将 i 值赋给 x3 后 i 值减 1*/
    x4=--i;                                 /*将 i 值减 1 后的结果赋给 x4*/
    printf("%d,%d,%d,%d\n",x1,x2,x3,x4);    /*输出 x1、x2、x3、x4*/
}
```

程序运行结果如图 6.6 所示。

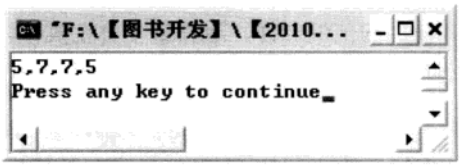


图 6.6 自加自减

使用自增自减操作符在进行运算时，是十分方便的，但是由于这种灵活性，也产生了程序的歧义和一些副作用。例如，i+++j 这个表达式是理解为（i++）+j，还是理解为 i+（++j）呢？C 语言的编译器在处理时会尽可能的按照从左到右将若干字符组成一个运算符，所以 i+++j，将被解释为（i++）+j，而不是 i+（++j）。如果需要按照 i+（++j）进行计算，则需要用括号将其分隔开。

所以，在书写多个运算符连接的表达式时，应该加上括号进行区分，这样既使程序脉络清晰，容易阅读，又避免出现错误。

细节 12 C 语言中的特殊表达式

例如，一个 C 语言的语句如下：

```
i=i+1;
```

可以将这一表达式改写成如下的形式：

```
i += 1;
```

这种表达式对于基本的算术运算都是有效的，所以可以将此类表达式写成下面的形式：

```
e1 op =e2;
```

其中，e1，e2 表示运算变量，op 表示的是运算符。它实际上就相当于如下表达式的形式：

```
e1 = (e1) op (e2);
```

注意：上述代码中 e2 表达式所带的括号是不可省略的。

表 6.2 所示的是基本表达式和特殊表达式之间的对应关系。

表 6.2　　基本表达式和特殊表达式之间的对应关系

基本表达式	特殊表达式
i = i + j;	i += j;
i = i - j;	i -= j;
i = i * j;	i *= j;
i = i / j;	i /= j;
i = i % j;	i %= j;

例如，下面的程序代码：

```
a *= c;
```

实际上执行的是：

```
a=a*c;
```

特别要注意等号后面的运算量是一个表达式的情况，如以下代码：

```
a *= c +d;
```

C 编译器执行的时候，会将上述表达式转换为下面的形式：

```
a = a * (c + d);
```

细节 13　特殊表达式 e1?e2:e3

在使用 if 语句的时候，我们经常会看到下面的代码：

```
if (a>b)
    c=a;
else
    c=b;
```

很显然，上面的语句是求最大值运算，如果大于，则将 a 的值赋给 c，否则将 b 的值赋给 c。在 C 语言中提供了一种特殊的表达式，使得上述代码得以简化，形式如下：

```
e1?e2:e3;
```

如果 e1 为真，则执行 e2，否则执行 e3。若想求两个数的最大值，则其命令的写法如图 6.7 所示。

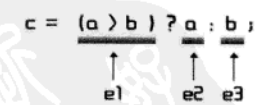


图 6.7　执行过程

细节 14　使用 printf 函数输出%

printf 函数的作用是向终端输出若干个任意类型的数据。printf 函数是一个标准库函数，它的函数原型在头文件“stdio.h”中。printf 函数调用的一般形式为：

```
printf(格式控制，输出表列)
```

其中“格式控制”是用双引号括起来的字符串，也称“转换控制字符串”，格式控制可由格式字符串和非格式字符串两种组成。格式字符串是以%开头的字符串，在%后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。

非格式字符串也就是通常所说的普通字符，即在输出时原样输出。例如：

```
printf("%d,%d",a,b);
```

这里的“%d”就是格式字符串；“,”是非格式字符串即普通字符；a 和 b 是输出表列。

使用 printf 输出%，只需重复百分号（使用%%）即可实现。强调用 printf 输出%是因为%是 printf 的转义字符。任何时候 printf 遇到%都会等待下一个字符，然后决定如何处理。而双%%就被定义成了单独的%字符。

注意：要想输出单个%字符时不用使用转义字符反斜杠\。

例 6.3 使用 printf 函数输出% (光盘位置：光盘\MR\Instance\6\03)。

例如：

```
#include<stdio.h>
main()
{
    int i,j;
    float r1,r2;
    printf("please input i and j:\n");
    scanf("%d,%d",&i,&j);                          /*输入变量 i 和 j 的值，注意中间用逗号隔开*/
    r1=i*0.3;
    r2=j*0.6;
    printf("%d*30%%=%f\n",i,r1);                   /*将 0.3 以百分号形式输出*/
    printf("%d*60%%=%f\n",j,r2);                   /*将 0.6 以百分号形式输出*/
}
```

程序运行结果如图 6.8 所示。

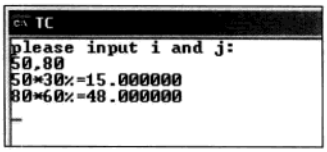


图 6.8 输出%

细节 15 %lf 与%f 差异

scanf 函数是一个标准库函数，它的函数原型在头文件“stdio.h”中，scanf 函数的一般形式为：

```
scanf(格式控制，地址表列);
```

其中，格式控制的作用与 printf 函数相同，但不能显示非格式字符串（普通字符），也就是不

能显示提示字符串。地址表列中给出各变量的地址。地址是由地址运算符“&”后跟变量名组成的。例如：&a 和&b 分别表示变量 a 和变量 b 的地址。这个地址就是编译系统在内存中给 a,b 变量分配的地址。

在 scanf 中用%lf 输入一个 double 类型的数据，而在 printf 中却不能使用%lf 输出 double 类型数据，只能使用%f 输出。

根据“默认参数提升”规则，在 printf 函数的可变参数列表中，不论作用域内有没有原型，都适用这一规则，float 型会被提升为 double 型。而对于 scanf，情况就完全不同了，没有类型的提升，所以 scanf 区别%f 和%lf。

例 6.4　类的封装（光盘位置：光盘\MR\Instance\6\04）。

程序代码如下：

```
#include<stdio.h>
main()
{
    float x;                                    /*定义 x 为单精度型*/
    double y;                                   /*定义 y 为双精度型*/
    printf("please input x and y:\n");
    scanf("%f,%lf",&x,&y);                      /*输入两个实型变量分别赋给 x 和 y*/
    printf("x is:%f\n",x);                      /*输出 x 的值*/
    printf("y is:%f",y);                        /*输出 y 的值*/
}
```

程序运行结果如图 6.9 所示。

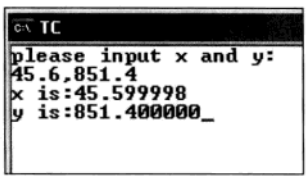


图 6.9　实行数据输出

细节 16　scanf 与 gets

例 6.5　scanf 与 gets(光盘位置：光盘\MR\Instance\6\05)。

在编写程序的过程中经常会遇到这种情况，如下代码：

```
#include<stdio.h>
main()
{
    int age;
    char name[30],adr[30];
    printf("please input age:\n");
    scanf("%d",&age);                                   /*输入年龄*/
    printf("please input name:\n");
```

```
    gets(name);                                          /*输入名字*/
    printf("please input adress:\n");
    gets(adr);                                           /*输如地址*/
    printf("age:%d\nname:%s\nadress:%s\n",age,name,adr);
}
```

在运行时会出现图 6.10 所示情况。

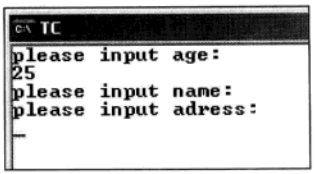


图 6.10 scanf 和 gets

上面的代码中用 scanf 读取一个数字，然后再用 gets()读取字符串，但是编译器好像跳过了第一个 gets()调用，直接到了第二个 gets()调用。为什么会出现这种情况呢？因为 scanf 不处理结尾的换行符。如果输入的数字后边紧接着一个换行符，则换行符会被 gets()处理。作为一般规则，不能混用 scanf()和 gets()，或任何其他的输入例程的调用。scanf 对换行符的特殊处理几乎一定会带来问题，所以要用 scanf 时就只用 scanf，这样就不会出现上述问题。

细节 17 什么时候用 switch

什么时候用一条 switch 语句比用多条 if 语句更好?

如果有两个以上基于同一个数字型变量的条件表达式，那么最好使用一条 switch 语句。与其使用如下代码：

```
if (x ==1)
    printf ("A\n");
else if (x ==2)
    printf ("B\n");
else if (x ==3)
    printf ("C\n");
else
    printf("D~Z\n");
```

不如使用 switch 语句编写的代码，它更易于阅读和维护。

```
switch (x)                                   /*根据 x 的值进行不同操作*/
{
case 1:
    printf ("A\n");                          /*输出 A*/
    break;
case 2:
    printf ("B\n");                          /*输出 B*/
    break
case 3:
    printf("C");                             /*输出 C*/
    break;
default:
    printf("D~E\n");                         /*输出 D~E*/
    break;
}
```

注意： 使用 switch 语句的前提是条件表达式必须基于同一个数字型变量。例如，尽管下述 if 语句包含两个以上的条件，但该此程序代码不能使用 switch 语句，因为这段程序是关于字符串比较的，而不是数字比较。

```
char *string="bibi";
if(!strcmp(name, "cici"))                          /*判断 name 是否是 cici*/
    printf("not cici");
else if(!strcmp(name, "didi"))                     /*判断 name 是否是 didi*/
    printf("not didi");
else if(!strcmp(name, "nini"))                     /*判断 name 是否是 nini*/
    printf("not nini");
else
    printf("probably aiai or bibi");
```

然后再强调一个许多人比较关心的效率的问题，即 switch…case 语句和 if…else if 语句哪个效率更高呢？

比较 switch…case 和 if…else if 的具体效率是不准确的。效率的高低关键要看实际代码，少数情况下，if…else if 写法得当的话，效率是高于 switch…case 的。

如以下代码：

```
void Test()
{
unsigned int x ;
        srand((unsigned long)time(0));             /*以系统时间为种子*/
        x=rand()%100;                              /*产生 100 以内的随机数*/
        if(x==0)
        {
            …
        }
        else if(x==1)
        {
            …
        }
        else
        {
            …
        }
}
```

x 为 0～100 范围内的随机数，当 x 不为 0 和 1 时执行 else 分支，因此上面函数有 98%的概率是执行 else 分支的，而执行 if 分支和 else if 分支的概率各为 1%。在执行 else 分支时，要先执行 if 判断语句，再执行 else if 判断语句后才会执行 else 分支里的内容。因此执行 else 分支需要进行两次判断。如果我们按下列方式将 else 分支的执行改成放到 if 里面去执行，则效率将更加提高。

```
void Test()
{
unsigned int x;
        srand((unsigned long)time(0));             /*以系统时间为种子*/
        x=rand()%100;                              /*随机产生 100 以内的数*/
        if(x>1)
        {
            ……
        }
        else if(x==0)
        {
            ……
        }
        else
```

```
        {
            ……
        }
}
```

如果上面的代码用 switch...case 来实现，则效率不如上面。一个好的程序，不仅要看它实现的功能如何，还要看其效率高低。

细节 18 if 语句的多个判断问题

首先来看两个小程序，分析一下这两个程序中哪个程序的写法更好，好的原因是什么？

程序 1：

```
#include<stdio.h>
main()
{
    int i,j,k;
    printf("pleasse input:\n");
    scanf("%d,%d,%d",&i,&j,&k);
    if(i>j||i>k)                                    /*i 大于 j 或 i 大于 k 有一个成立则执行下面语句*/
    {
        printf("i=%d is not the minimal number!",i);  /*输出最小数*/
        exit(0);
    }
    ……
}
```

程序 2：

```
#include<stdio.h>
main()
{
    int i,j,k;
    printf("pleasse input:\n");
    scanf("%d,%d,%d",&i,&j,&k);                     /*输入三个数，分别赋给 i、j、k*/
    if(i>j)                                         /*判断 i 是否大于 j*/
    {
        printf("i=%d is not the minimal number!",i);
        exit(0);
    }
    if(i>k)                                         /*判断 i 是否大于 k*/
    {
        printf("i=%d is not the minimal number!",i);
        exit(0);
    }
    ……
}
```

答案是程序 1 的写法更优。为什么呢？因为程序 1 写法中少了语句：

```
{
    printf("i=%d is not the minimal number!",i);
    exit(0);
}
```

编译后，执行代码会比第一种写法小，执行时耗用的内存自然少了，其他方面的性能都是一

样的。

归结一下，在 if 语句中能使用逻辑运算符来减少 if 语句使用数量时，要尽量使用逻辑运算符。这样可以减少重复代码，使程序更加简单明了。

细节 19　break 在 switch 语句中的使用

例 6.6　break 在 switch 语句中的使用(光盘位置：光盘\MR\Instance\6\06)。

在使用 switch 循环的时候，经常要使用到 break 语句，如下：

```
main()
{
    int score;
    printf("\nplease enter score(score<=100):");
    scanf("%d", &score);                                    /*输入学生成绩*/
    if (score == 100)                                       /*如果成绩是 100 则和 90 是一样的等级*/
        score = 90;
    score = score / 10;                                     /*求出成绩的高位数字*/
    switch (score)
    {
    case 9:
        printf("the grade is A");                           /*等级为 A*/
        break;
    case 8:
        printf("the grade is B");                           /*等级为 B*/
        break;
    case 7:
        printf("the grade is C");                           /*等级为 C*/
        break;
    case 6:
        printf("the grade is D");                           /*等级为 D*/
    }
}
```

当输入 65 时，程序运行结果如图 6.11 所示。

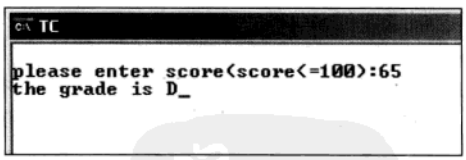


图 6.11　成绩评定

通过上面的程序代码可以发现前三个 case 语句后都会有 break 语句，而最后一个 case 语句后面没有出现 break，是不是 switch 最后面的语句（case 或 default）一定不需要跟 break 语句呢？switch 语句的最后一个分支通常是不需要 break 语句的，但是建议编写程序的时候最好还是在 switch 语句的每个分支后面加上 break 语句，包括最后一个分支。主要原因是：一个人编写的程序很可能要让另一个人来维护，而这个人很有可能要增加一些新的分支来完善一下程序的功能，如果没有注意到之前最后一个分支是否有写 break 语句，结果会使原来的最后一个分支受到其后新增分支的干扰而失去原有的功能。若改写上面代码，形式如下：

```
main()
{
    int score;
    printf("\nplease enter score(score<=100):");
    scanf("%d", &score);                                    /*输入学生成绩*/
    if (score == 100)                                       /*如果成绩是 100 则和 90 是一样的等级*/
```

```
        score = 90;
    score = score / 10;                          /*求出成绩的高位数字*/
    switch (score)
    {
    case 9:
        printf("the grade is A");                /*等级为 A*/
        break;
    case 8:
        printf("the grade is B");                /*等级为 B*/
        break;
    case 7:
        printf("the grade is C");                /*等级为 C*/
        break;
    case 6:
        printf("the grade is D");                /*等级为 D*/
    case 5:
        printf("\nmakeup");
        break;
    }
}
```

如果此时再从键盘中输入 65，则运行结果如图 6.12 所示。

本意是希望在输入 65 时，只输出“the grade is D”，而不输出“makeup!!”。出现以上情况的原因是没有在源程序的最后一个 case 语句后面加 break。在每个分支后面加上 break 语句将防止发生这种错误并增强程序的安全性。此外，目前大多数优化编译程序都会忽略最后一条 break 语句，所以加入这条语句不会影响程序的性能。

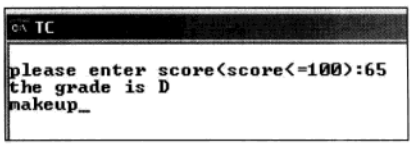


图 6.12 break 作用

细节 20 while 与 do...while 的异同

while 语句和 do…while 语句类似，都是要判断循环条件是否为真，如果为真则执行循环体，否则退出循环。

while 语句和 do…while 语句的区别在于 do…while 语句是先执行一次循环体，然后再判断。因此 do…while 语句至少要执行一次循环体；而 while 是先判断后执行，如果条件不成立不满足，则一次循环体也不执行。

例 6.7 do…while 和 while 区别（光盘位置：光盘\MR\Instance\6\07）。

程序运行结果如图 6.13 所示。

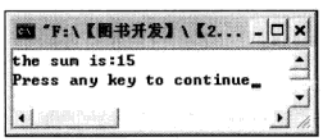


图 6.13 do...while 求和

程序代码如下：

```
#include<stdio.h>
main()
{
    int i=15,sum=0;
    do                                          /*使用 do-while 判断*/
    {
        sum=sum+i;                              /*循环体语句*/
    }while(i<15);
    printf("the sum is:%d\n",sum);
}
```

通过上面两个例子会将 while 语句换成 do...while 语句，程序运行出的结果就截然不同，主要是因为 while 语句先判断后执行，先判断 i<15 表达式是否为真，因为表达式为假，故没有执行循环体语句，而 do-while 是先执行后判断，无论表达式是否为真都先执行，执行完再判断。

while 语句和 do-while 语句一般都可以互相改写，但是 do-while 语句和 while 语句在相互替换时，要注意修改循环控制条件。

细节 21　while、do...while、for 循环相互转换

将下面用 while 语句写的程序分别用 do...while 和 for 与语句来实现。

```
main()
{
    int i=2,n;                                  /*定义变量 i，n 为基本整型并为 i 赋初值 2*/
    long fac=1;                                 /*定义 fac 为单精度型并赋初值 1*/
    printf("please input an interger>=0.\n");
    scanf("%d",&n);                             /*使用 scanf 函数获取 n 的值*/
    if(n==0||n==1)                              /*当 n 为 0 或 1 的时候输出阶层为 1*/
    {
        printf("factorial is 1.\n");
        return 0;
    }
    while(i<=n)                                 /*当满足输入的数值大于等于 i 时执行循环体语句*/
    {
        fac=fac*i;                              /*实现求阶层的过程*/
        i++;                                    /*变量 i 自加*/
    }
    printf("factorial of %d is:%ld.\n",n,fac);  /*输出 n 和 fac 最终的值*/
}
```

改写成 do...while 语句，代码如下：

```
main()
{
    int i=2,n;                                  /*定义变量 i，n 为基本整型并为 i 赋初值 2*/
    long fac=1;                                 /*定义 fac 为单精度型并赋初值 1*/
    printf("please input an interger>=0.\n");
    scanf("%d",&n);                             /*使用 scanf 函数获取 n 的值*/
    if(n==0||n==1)                              /*当 n 为 0 或 1 的时候输出阶层为 1*/
    {
        printf("factorial is 1.\n");
        return 0;
    }
```

```
    do
    {
        fac=fac*i;                                  /*实现求阶层的过程*/
        i++;                                        /*变量 i 自加*/
    }while(i<=n);
    printf("factorial of %d is:%ld.\n",n,fac);      /*输出 n 和 fac 最终的值*/
}
```

改写成 for 语句，代码如下：

```
main()
{
    int i,n;                                        /*定义变量 i，n 为基本整型并为 i 赋初值 2*/
    long fac=1;                                     /*定义 fac 为单精度型并赋初值 1*/
    printf("please input an interger>=0.\n");
    scanf("%d",&n);                                 /*使用 scanf 函数获取 n 的值*/
    if(n==0||n==1)                                  /*当 n 为 0 或 1 的时候输出阶层为 1*/
    {
        printf("factorial is 1.\n");
        return 0;
    }
    for(i=2;i<=n;i++)                               /*当满足输入的数值大于等于 i 时执行循环体语句*/
        fac=fac*i;                                  /*实现求阶层的过程*/
    printf("factorial of %d is:%ld.\n",n,fac);      /*输出 n 和 fac 最终的值*/
}
```

细节 22 循环不能交叉

跟其他的各种编程语言一样，循环控制语句也允许有双层嵌套循环语句的存在。所谓的循环嵌套控制结构就是在一个循环结构当中包含另外一个循环结构，基本形式如图 6.14 所示。

如果用图 6.15 所示的符号来表示循环。

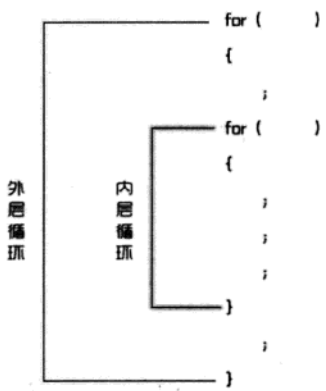


图 6.14 循环嵌套



图 6.15 循环符号

则图 6.16 中所有的复杂的循环都是允许的。

在使用循环嵌套的时候，需要注意的是，循环不可以有交叉的情况，即两个循环不可能有交叉

产生，例如像图 6.17 所示的循环交叉就是不允许的。

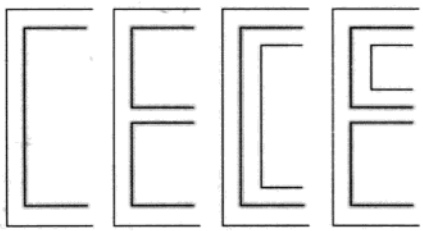


图 6.16　各种复杂的循环嵌套

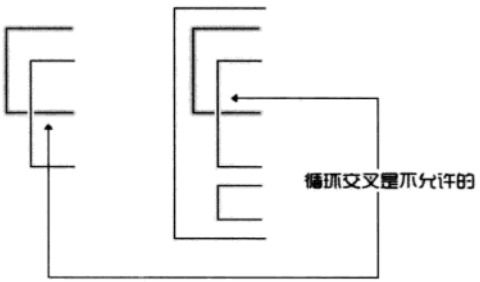


图 6.17　循环交叉是不允许的

细节 23　goto 语句使用规则

使用 goto 语句时有以下三点要注意。

- 使用 goto 语句只能 goto 到同一函数内，而不能从一个函数里 goto 到另外一个函数里。
- 使用 goto 语句在同一函数内进行 goto 时，goto 的起点应是函数内一段小功能的结束处，goto 的目的 label 处应是函数内另外一段小功能的开始处。
- 不能从一段复杂的执行状态中的位置 goto 到另外一个位置，比如：不可以从多重嵌套的循环判断中跳出去。

细节 24　合理使用 return 语句

return 语句将被调用函数中一个确定值带回主调函数中。如果需要从被调函数返回一个函数值，被调函数中必须包含 return 语句。当函数执行到 return 语句时，就按照 return 关键字后面的要求返回相应的内容给主调函数。即使被调函数后面还有语句也不再执行。因此 return 语句也可以看作是函数执行的结束语句。如果不需要从被调函数中返回值，则可以不使用 return 语句。

例如：

```
int fuc(int x)
{
    if x>=0
        return 1;
    else
        return 0;
}
```

这是一个自定义的函数，该函数的作用是判断 x 的值的正负，如果 x 的值大于等于 0，则函数返回 1；如果 x 小于 0，则返回 0。在这个函数中有两个 return 语句，分别是 if 语句的两个分支。当执行完其中任何一个 return 语句以后就不再执行了，该函数也就执行完毕了。

说明：（1）一个函数中可以有一个以上的 return 语句，执行到哪个 return 语句，哪个语句就起作用。

（2）return 语句后面可以使用括号，也可以不使用括号。例如：

return x;

也可以写成：

return (x);

（3）return 语句后面可以是一个值也可以是一个表达式。例如：

return 1;

return x;

return a+b;

这些用法都是正确的用法。

细节 25 合理使用空格使代码更规范

（1）关键字之后要留一个空格。像 const、case 等关键字之后要保留一个空格，否则编译器无法辨析它是关键字。if、for、while 等关键字之后应该留一个空格，然后再跟小括号“(”，以突出显示关键字。

（2）在函数名之后不要留空格，要紧跟“(”，以示与关键字的区别。

（3）“(”后向紧跟，“)”、“,”、“;”前向紧跟，紧跟处不留空格。

（4）“,”之后要留空格，如果“;”不是一行的结束，则最后要留空格。

（5）赋值操作符、比较操作符、算术操作符、逻辑操作符、位操作符。如“=”、“+=”、“>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”、“^”等二元操作符的前后都应该适当加空格。对于比较长的表达式，即使是使用了这些二元操作符，也应该适当地去掉一些空格，使表达式看起来紧凑。

（6）一元操作符如“!”、“~”、“++”、“—”等前后不加空格。像“[]”、“.”、“->”等操作符，同样前后不加空格。

例如下面的代码，属于良好的编程风格。

```
if (a>=2000)
for (i=0; i<10; i++)
void fun(int a);
arr[5]=1;
a.fun();
b->fun();
```

细节 26 C 语言如何解释 a<b<c

首先来看下面的代码，输出结果是什么？

```
int main(void)
{
    int a=1,b=2,c=2;
    while(a<b<c)
    {
        int t;
        t=a;
        a=b;
        b=t;
        c--;
    }
    printf("%d\n",c);
    return 0;
}
```

这段代码很有欺骗性，有些开发人员会觉得循环不会执行，结果应该是 2，可是编译运行以后，得到如图 6.18 所示的结果，结果是 0。

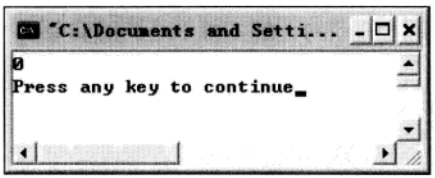


图 6.18　运行结果图

产生如图 6.18 所示的程序运行结果的原因是：a<b<c 被解释成（a<b）<c，如果（a<b）成立，则（a<b）等于 1，否则（a<b）等于 0。



第7章 19个C语言重点知识技术细节

数组和指针是 C 语言程序开发中的重点知识。当处理的同类数据的量比较大时，应用数组会简化这些繁琐的代码。

C 语言的精髓和核心所在就是指针，用好了指针可以在 C 语言编程中起到事半功倍的效果。

希望读者可以通过对这些技术细节的了解，可以达到对 C 语言应用自如。

细节 01 判断数组中元素的个数

一维数组是用以存储一维数列中数据的集合。一维数组的定义方式如下：

```
类型说明符 数组名[常量表达式];
```

- 数组名就是这个数组型变量的名字，命名规则与变量名一致。
- 数组中存储元素的数据类型由类型说明符给出，可以是任意的数据类型（整型、实型、字符型等）。
- 常量表达式定义了数组中存放的数据元素的个数，即数组长度。例如 a[5]，5 表示数组中有 5 个元素，下标从 0 开始，到 4 结束。

注意：在数组 a[5]中只能使用 a[0]、a[1]、a[2]、a[3]、a[4]，而不能使用 a[5]，若使用 a[5]会出现下标越界的错误。

在编写程序过程中，往往需要知道一个数组中有多少个元素，那么怎么才能判断出一个数组中有多少个元素呢？可以采用下面这个方法：

```
int a[]={1,2,5,3,6,9,45,6,6,59,54,64,63,563,5,85};
int n;
n=sizeof(a)/sizeof(a[0]);
```

上面代码是对一个基本整型数组而言的，对于一个字符型数组或者其他类型数组也是一样的，可以套用此方法。

例如：

```
#include<stdio.h>
main()
{
    float a[]={32.3,12.3,2.3,0.3,6.3,53.9,45.2,1.36,9.26,23.52,31,4.363};        /*给数组赋值*/
    int n;
    printf("the length of array a is:\n");
    n=sizeof(a)/sizeof(a[0]);                                                    /*获取数组中元素个数*/
    printf("%d",n);
}
```

运行结果如图 7.1 所示。

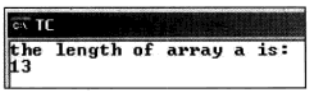


图 7.1　求数组长度

细节 02　向函数传递二维数组

在编写程序的过程中可以用数组名作为函数参数，这种方法实际上是通过数组的首地址传递整个数组。

如何向一个函数中传递二维数组？数组与函数分别如下：

```
int array [row][col];
f(array);
```

要想实现传递很关键一点就是函数的声明必须匹配，如下：

```
void f(int a[][col])
{
......
}
```

这里编译器进行了通常的从“数组的数组”到“数组的指针”的转换。

或者：

```
void f(int (*p)[col])
{
......
}
```

这种形式中采用了指针，因为被调用的函数并不为数组分配地址，所以它并不需要知道数组的大小，所以行的大小可以省略，但列的大小不能省。

细节 03　字符数组的结束标志

在 C 语言中，使用字符数组保存字符串，也就使用一个一维数组保存字符串中的每一个字符。此时系统会自动为其添加'\0'作为结束符。

例如在初始化一个字符数组时：

```
char cArray[]="Hello";
```

字符串总是以'\0'作为串的结束符。因此当把一个字符串存入一个数组时，也把结束符'\0'存入数组，并以此作为该字符串是否结束的标志。

注意：有了'\0'标志后，字符数组的长度就显得不那么重要。当然在定义字符数组是应估计实际字符串长度，保证数组长度始终大于字符串实际长度。如果在一个字符数组中先后存放多个不同长度的字符串，则应使数组长度大于最长的字符串的长度。

用字符串方式赋值比用字符逐个赋值要多占一个字节，多占的这个字节用于存放字符串结束标志'\0'。上面的字符数组 cArra 在内存中的实际存放情况如图 7.2 所示。

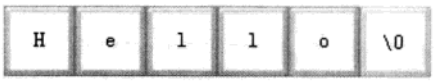


图 7.2　内存中存储情况

'\0'是由 C 编译系统自动加上的。所以上面的赋值语句等价于：

```
char cArray[]={'H','e','l','l','o','\0'};
```

字符数组并不要求最后一个字符为'\0'，甚至不包含'\0'也可以。像下面这样写也是合法的：

```
char cArray[5]={'H','e','l','l','o'};
```

不过是否需要加'\0'，完全根据需要决定。但是由于系统对字符串常量自动加一个'\0'。因此，为了使处理方法一致，便于测定字符串的实际长度，以及在程序中作相应的处理，在字符数组也常常人为的加上一个'\0'。例如：

```
char cArray[6]={'H','e','l','l','o','\0'};
```

细节 04　extern 声明数组

由于 C 语言允许将一个较大的程序分成若干独立模块文件分别编译，如果一个源文件中的函数想引用其他源文件中的变量，就可以用 extern 来声明外部变量。这就是说 extern 变量可以扩展外部变量的作用域。

定义时默认 static 关键字的外部变量，即为非静态外部变量。其他源文件中的函数引用非静态外部变量时，需要在引用函数所在的源文件中进行说明。

```
extern  数据类型  数组;
```

注意：在函数内的 extern 变量说明，表示引用本源文件中的外部变量，而函数外（通常在文件开头）的 extern 变量说明，表示引用其他文件中的外部变量。

如果一个程序代码如下：

```
#include<stdio.h>
main()
{
    void func(int x,int y);                          /*函数声明*/
    extern int a[5];                                 /*extern 声明数组*/
    int m=7,n=9,len;                                 /*定义变量并为部分变量赋值*/
    printf("%d\n",a[2]);
    len=sizeof(a)/sizeof(a[0]);                      /*获得数组元素的个数*/
    func(m,n);                                       /*调用函数 func*/
    printf("length:%d",len);                         /*输出数组长度*/
}
int a[5]={3,5,6,4,2};                                /*定义数组*/
void func(int x,int y)                               /*自定义函数 func*/
{
    if(x>y)
    {
        x=a[0];
        y=a[4];
    }
    else
    {
        x=a[3];
        y=a[1];
    }
    printf("x=%d,y=%d\n",x,y);                       /*输出 x，y 的值*/
}
```

则其运行结果如图 7.3 所示。

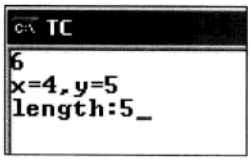


图 7.3　extern 声明数组

如果将程序改写成如下形式：

```
#include<stdio.h>
main()
{
    void func(int x,int y);                          /*函数声明*/
    extern int a[];                                  /*extern 声明数组，不指定数组长度*/
    int m=7,n=9,len;                                 /*定义变量，并为部分变量赋初值*/
    printf("%d\n",a[2]);                             /*输出第三个元素*/
    len=sizeof(a)/sizeof(a[0]);                      /*获取数组的长度*/
    func(m,n);                                       /*调用函数 func*/
    printf("length:%d",len);                         /*输出长度*/
}
int a[]={3,5,6,4,2};                                 /*定义数组*/
void func(int x,int y)                               /*定义 func 函数*/
{
    if(x>y)
    {
        x=a[0];
        y=a[4];
```

```
    }
    else
    {
        x=a[3];
        y=a[1];
    }
    printf("x=%d,y=%d\n",x,y);                    /*输出 x 和 y 的值*/
}
```

则会提示如图 7.4 所示的错误信息。

```
Message
Compiling E:\CPRO\F2.C:
•Error E:\CPRO\F2.C 9: Size of structure or array not known in function main
```

图 7.4　错误信息提示

这两个程序不同之处就在于后面的程序没有指定数组的长度，所以不能使用 sizeof 函数，因为 sizeof 在编译时发生作用，它不能获得后面定义的数组的大小，但是如果 sizeof 用在定义的数组的后面，即使不指定数组大小也能获取其长度，程序代码如下：

```
#include<stdio.h>
main()
{
    extern int a[];                               /*声明 extern 输出，不指定长度*/
    void func(int x,int y);                       /*声明 func 函数*/
    int m=7,n=9;
    printf("%d\n",a[2]);                          /*输出第三个元素的值*/
    func(m,n);
}
int a[]={3,5,6,4,2};                              /*定义数组*/
void func(int x,int y)                            /*定义数组*/
{
    int len;                                      /*定义变量*/
    len=sizeof(a)/sizeof(a[0]);                   /*获取数组元素个数即数组长度*/
    printf("length:%d\n",len);
    if(x>y)
    {
        x=a[0];
        y=a[4];
    }
    else
    {
        x=a[3];
        y=a[1];
    }
    printf("x=%d,y=%d\n",x,y);                    /*输出 x 和 y 的值*/
}
```

还可以在数组的最后一个元素放入“哨兵”值（通常是 0，-1，或者 NULL），这样代码不需要数组大小也可以确定数组的长度，代码改写如下：

```
#include<stdio.h>
main()
{
    extern int a[];                               /*声明 extern 数组，不指定长度*/
    void func(int x,int y);                       /*声明函数*/
    int m=7,n=9;
    printf("%d\n",a[2]);
```

```
    func(m,n);                                    /*调用 func 函数*/
}
int a[]={3,5,6,4,2,-1};                           /*定义数组*/
void func(int x,int y)                            /*定义 func 函数*/
{
    int len;
    len=sizeof(a)/sizeof(a[0]);                   /*获取数组长度*/
    printf("length:%d\n",len);
    if(x>y)
    {
        x=a[0];
        y=a[4];
    }
    else
    {
        x=a[3];
        y=a[1];
    }
    printf("x=%d,y=%d\n",x,y);                    /*输出 x 和 y 的值*/
}
```

细节 05　数组下标从 0 开始

数组元素在数组中顺序排列编号，这些元素的编号称为数组元素的下标，由于有数组下标，数组元素在数组中的位置就被唯一的确定下来。

在 C 语言中规定数组下标必须从 0 开始，对于数组 a[n]，说明数组 a 有 n 个元素，其中 n 必须为正数，数组的下标范围为 0～n-1，即数组的最大下标为 n-1，最小下标为 0。

细节 06　数组名作函数参数与数组元素做函数参数的不同之处

用数组元素做实参时，只要数组类型与函数的形参类型一致，那么作为下标变量的数组元素的类型也和函数形参变量的类型是一致的。

数组名就是数组的首地址。因此在数组名作为函数的参数时，所进行的传送只是地址的传送，即把实参数组的首地址赋予形参数组名。

用数组名作为函数的参数时还应该注意以下几点。

（1）形参数组和实参数组的类型必须是一致的，否则将引起错误。

（2）形参数组和实参数组的长度可以不相同，在参数传递时，只传送首地址而不检查形参数组的长度。

（3）在函数形参表中，允许不给出形参数组的长度或用一个变量来表示数组元素的个数。

（4）多维数组也可以作为函数的参数。在函数定义时对形参数组可以指定一位数组的长度，也

可以省去一维的长度。

细节 07 ‘\0’ 和 “\0” 的区别

字符串就是一串字符文字集合在一起，用字符数组来存储的一组字符，这时字符数组不是字符串。在 C 语言中没有字符串变量，字符串不是存放在一个变量中，而是存放在一个字符数组中。

‘\0’ 是 ASCII 码中值为 0 的字符，是一个 “空操作” 字符，不是可显示的字符，不进行任何操作，只是作为一个标记。

“\0” 是 C 语言规定的存放字符串的结束标志，占用 1B 内存空间。

例如，char c[] = “mrsoft”;

字符串的长度为 7，系统自动在末尾加 “\0”，因此在用字符串赋初值时，一般无需指定数组的长度。

细节 08 静态全局变量与全局变量

全局变量的说明之前再加上 static 就构成了静态的全局变量。全局变量也是通常所说的外部变量，其本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。这两者的区别在于作用域不同，非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。

那么什么时候使用静态全局变量呢？

因为静态全局变量的作用域局限于一个源文件内，只能被该源文件内的函数公用，所以当其他源文件中定义了相同名字的全局变量也没有关系，他们之间互不干扰，都只在各自的源文件中起作用。因此可以避免在其他源文件中引起错误。

细节 09 访问寄存器变量地址

通常变量的值存放在内存中，当对一个变量频繁读写时，则需要反复访问内存储器，从而花费大量的存取时间。为了提高效率，C 语言提供了另一种变量，即寄存器变量。这种变量允许将局部变量的值存放在 CPU 中的寄存器中，使用时，不需要访问内存，而直接从寄存器中读写。寄存器变量的说明符是 register。

可不可以访问寄存器变量地址呢？这个问题具体与编译器的实现有关，建议不要声明全局变量为寄存器变量，即使是局部变量都最好不要声明其为寄存器变量，现在的编译器在优化时都会较为合理的安排寄存器变量的使用，而人为的安排有时会造成优化的低效。

细节 10　参数校验

在 C 语言的函数中，一般都要对函数的参数进行校验，但是有些情况下不在函数内进行校验，而由调用者在外部校验，到底什么情况下应该在函数内进行校验，什么情况下不需要在函数内进行校验呢？

对于需要在大的循环里调用的函数，不需要在函数内对参数进行校验。

例如定义一个函数 List 用来遍历链表中的每个节点：

```
void *List (LIST *p)
```

在该函数中是否需要对参数 p 进行校验呢？

因为对链表中节点进行逐个遍历的函数通常是要在一个循环里使用，比如在一个要循环 1000 次的循环体语句里调用了该函数，也就是说要对一个有 1000 个节点的链表进行遍历。如果上面的函数对参数 p 进行了校验，那么对整个链表中的每个节点逐个遍历，这样就将校验 1000 次，不如由调用者在调用函数前校验一次就够了。因此，像这种可能频繁地被调用，且在外面校验只要校验一次就够的函数参数是不需要在函数内部进行校验的。

调用次数比较多的一般不校验。

对于调用次数较少的一般对参数要校验。

要从整个程序的执行时间上来看是否进行参数校验，在参数校验相对整个函数来讲占的比例可以忽略不计的情况下，一般最好对函数参数进行校验。因为进行参数校验可使程序更稳定。

细节 11　什么是空指针

空指针不会指向任何地方，它不是任何变量或函数的地址。取地址操作符&永远不会返回空指针。使用 malloc 函数分配内存空间时，如果成功，则不会返回空指针，否则会返回一个空指针。

空指针在概念上不同于未初始化的指针。空指针可以确保不指向任何地方，而未初始化的指针则可能指向任何地方。

在指针和链表中使用的非常广泛的空指针，表示的是不指向任何对象的一种指针。空指针通常定义一个常量 NULL 来表示，NULL 的值为 0。他是在头文件<stddef.h>中定义的一个宏，其值与任何有效的指针的值是不同的。NULL 是 0，它可能被强制转换为 void *或者 char *。即 NULL 可能是 0 或者（void *）0。

空指针一般应用于以下的情况。

- 用空指针终止对递归数组结构的间接引用；

- 用空指针作为函数调用失败时的返回值；
- 用空指针作为警戒值。

注意：不能间接引用一个空指针，否则，程序可能会得到一个毫无疑义的结果或者一个全是 0 的值而突然停止。

细节 12 if（p）检查指针是否为空

在 C 语言中计算表达式中的布尔值的时候，如果表达式等于 0 则认为表达式为假，否则为真。

例如：

```
if ( express )
```

无论 express 是何种表达式，对与编译器来说他都等价于：

```
if((express)!=0)
```

如果用指针 p 代替 express，则：

```
if(p)
```

等价于：

```
if(p!=0)
```

编译器是可以识别出 0 实际上是一个空指针常量，并使用正确的空指针值。按照上面的逻辑也可以出土对布尔求反操作：

```
if ( !express )
```

等价于：

```
if((express)==0)
```

如果用指针 p 代替 express，则：

```
if(!p)
```

等价于：

```
if(p==0)
```

所以说像类似于 if（p）这样的写法是完全合法的。

细节 13 指针变量赋值相关

在编写程序的过程中经常会出现如下错误：

```
int a=25;
int *p;
*p=&a;
```

又如：

```
int *p;
*p=malloc(16);
```

上述的这两种写法均是错误的，因为声明的是指针变量 p，而不是*p，当操作指针本身时，只需要使用指针变量本身，所以可将上面的代码改写成如下形式：

```
int a=25;
int *p;
p=&a;
```

和

```
int *p;
p=malloc(16);
```

去掉了 p 前面的“*”便是合法的形式，除了这种方法外，又有一种写法也是正确的，如下所示：

```
int a=25;
int *p=&a;
```

和

```
int *p=malloc(16);
```

细节 14　const 定义指针变量

分析：

```
char * const p;
char const * p;
const char *p;
```

上述三个语句有什么区别？

```
char * const p;
```

含义是常量指针，p 的值不可以修改。

```
char const * p;
```

含义是指向常量的指针，指向的常量值不可以修改。

```
const char *p;
```

含义和 char const *p 含义一样，都是指向常量的指针，指向的常量值不可以修改。

细节 15 char a[]与 char *a

学完数组再学指针，很多人会认为 char []与 char *a 是相同的，那么它们是否相同？

char []与 char *a 不是相同的，数组不是指针。定义一个字符型的数组 a[6]，此时要求内存中分配一个以名称 a 开始的 6 个字符的位置。而 char *b 是请求一个位置放置一个指针，这个指针可以指向任何位置，或者哪也不指向。

例如：

```
char a[]="hello";
char *p="mingri";
```

可分别用图 7.5 和图 7.6 表示。

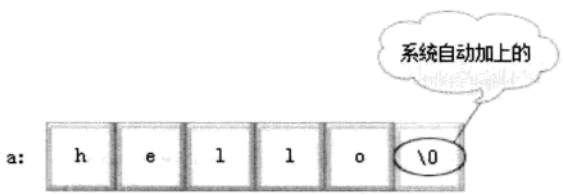


图 7.5　字符型数组存储

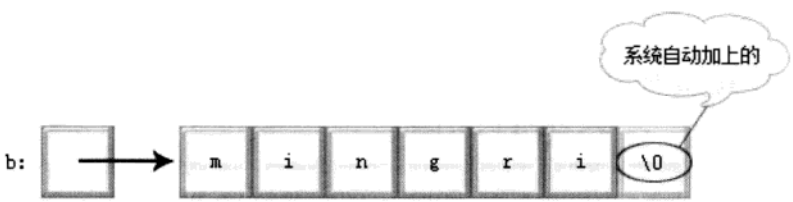


图 7.6　字符型指针

在学习 C 语言过程中常会听到“指针与数组等价”这句话，注意这里说得“等价”并不表示它们相同。实际上 C 语言中指针算术和数组下标运算等价，指针和数组是不同的。指针算术和数组下标等价可以用下面这段程序代码运行出的最终结果来体现。

```
#include<stdio.h>
main()
{
    char a[]="hello";
    char *b="mingri";
    printf("%c\n",a[4]);                    /*输出第 5 个字符*/
    printf("%c",b[4]);                      /*输出指针指向的第五个字符*/
}
```

运行的结果如图 7.7 所示。

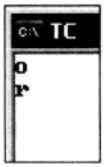


图 7.7　字符型数组与指针

细节 16　指针应用

分析下面给出的程序代码：

```
#include<stdio.h>
main()
{
    int a[5]={11,22,33,44,55};
    int *ptr=(int *)(&a+1);
    printf("%d,%d",*(a+1),*(ptr-1));                    /*输出指针变量指向的元素*/
}
```

程序运行运行结果如图 7.8 所示。

&a+1 不是简单的首地址加 1，系统会认为加一个 a 数组的偏移，是偏移了一个数组的大小，如图 7.9 所示。

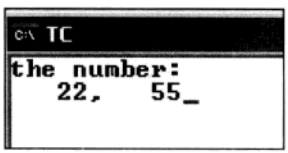


图 7.8　指针应用

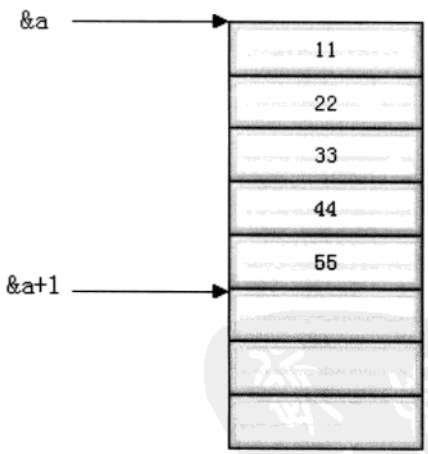


图 7.9　偏移

观察图 7.9 会发现本题中的偏移一个数组的大小也就是 5 个 int 的大小，如果数组 a 中含 10 个 int 元素，则&a+1 就是偏移了 10 个 int 的大小。

程序中：

```
int *ptr=(int *)(&a+1);
```

意思是将指针 ptr 指向图 7.8 中&a+1 的位置，而 ptr-1 则是将 ptr 减去一个 int 的大小，如图 7.10 所示。

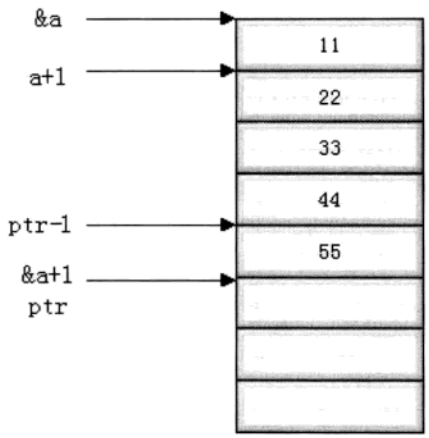


图 7.10　指针指向数组情况

总结如下。

&a 是数组指针，而指针加 1 要根据指针类型加上一定的值，不同类型的指针+1 之后增加的大小不同，a 是长度为 5 的 int 数组指针，所以要加 5*sizeof(int)。如果是长度为 10 的 int 数组指针，则要加上 10*sizeof（int），而 ptr 则是一个 int 型的指针变量，所以加 1 减 1，变化的只是一个 int 的大小，这有别于数组指针。

a 和&a 的地址是一样的，但意思不一样，a 是数组首地址，也就是 a[0]的地址，&a 是对象首地址，这里的对象指的就是 a 这个数组，a+1 是数组下一元素的地址，即 a[1]。&a+1 是下一个对象的地址，即 a[5]。

细节 17　指针类型和指针所指向的类型

指针类型和指针所指向的类型是两个概念。从语法角度看，只要把指针声明语句里的指针名称去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。

例如：

```
int *p;                    指针的类型是 int *。
char *p;                   指针的类型是 char *。
int **p;                   指针的类型是 int **。
int (*p)[5];               指针的类型是 int (*)[5]。
int *(*P)[5];              指针的类型是 int *(*)[5]。
```

用户通过指针来访问指针所指向的内存区域，指针所指向的类型决定了编译器将把此内存区域当做什么来看待。

从语法上看，只需将指针声明语句中的指针名称和名称左边的指针声明符*去掉，剩余部分就是指针所指向的类型。

例如：

```
int *p;                    指针所指向的类型是 int。
char *p;                   指针所指向的类型是 char。
int **p;                   指针所指向的类型是 int *。
int (*p) [5];              指针所指向的类型是 int ( ) [5]。
int *(*p) [5];             指针所指向的类型是 int * ( ) [5]。
```

细节 18　指针运算与取地址运算符的联系

指针运算符*表示“指向”，表示指针变量与它所指向的变量之间的联系。地址运算符&表示变量的地址，&a 表示变量 a 的地址。

使用*可以指向变量的地址，使用&可以获取变量的地址，可见*和&是互为逆运算的。假设 p 为一个指针变量，则可知：

- p 表示指针变量；
- *p 表示 p 所指向的变量；
- &p 表示指针变量的地址；
- &（*p）表示变量（*p）的地址，即 p；
- *（&p）表示（&p）所指向的变量，即 p。

细节 19　函数型指针

一个函数可以返回一个整型值、字符值、实型值等，同时一个函数的返回值也可以是指针型的数据，即地址。返回指针类型的数值时与前面讲过的其他数值型基本类似，含有指针型函数值的函数的声明一般为：

```
数据类型 *函数名（形参列表）
```

例如：

```
int *max(int x,int y)
```

max 是函数名，调用它以后能得到一个指向整形数据的指针。x 和 y 是函数 max 的形式参数，这两个参数也均为基本整型。这个函数的函数名前面有一个*，表示此函数是指针型函数，类型说明是 int 表示返回的指针指向整型变量。

例 7.1　使用返回指针的函数查找最大值。（光盘位置：光盘\MR\Instance\7\01）。

```
#include<stdio.h>
int *FindMax(int *p, int n)                          /*自定义函数，实现最大值的查找*/
{
    int i,*max;                                      /*定义 max 为指针变量，I 为基本整型变量*/
    max = p;
    for (i = 0; i < n; i++)
        if (*(p + i) >  *max)
            max = p + i;                             /*把最大数的地址赋给变量*/
        return max;                                  /*将 max 的值返回*/
}
main()
{
    int a[10],*max, i;
    printf("Please input ten integer:\n");
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &a[i]);                          /*给数组 a 中各元素赋值*/
    }
    max = FindMax(a, 10);                            /*调用 FindMax 函数*/
    printf("The max number is: %d\n",  *max);        /*输出查找到的最大值*/
    getch();
}
```

程序运行结果如图 7.11 所示。

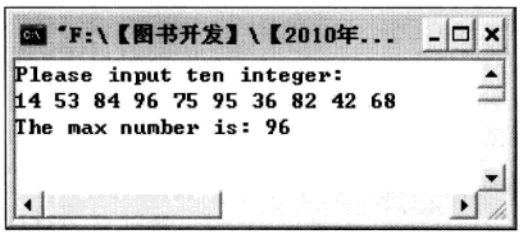


图 7.11　查找最大值

程序中自定义了一个返回指针值的函数：

```
int * fun(int x,int y)
```

该函数的作用是找到这个一维数组中最大的数，函数中定义了指针变量 max，max 初值为数组的首地址，将 max 的值与数组中的每个数据做比较，如果数组中的元素大于 max 指向的元素，则将该元素的值赋给 max，依此类推，直到整个一维数组中的元素均比较完毕，此时 max 所指向的值就是这个一维数组中最大的元素。在主函数中定义了一个指针变量 max，该变量 max 的值是函数 FindMax 的返回值，即一个地址，通过访问该地址可以获得这个一维数组的最大值。



第8章　18个C语言高级技术细节

C语言高级应用包括预处理和内存管理等。预处理功能是C语言特有的功能。可以使用预处理和具有预处理的功能是C语言与其他高级语言的区别之一。

程序在运行时，将需要的数据都组织存放在内存空间，以备程序使用。在软件开发的过程中，常常需要动态地分配和撤销内存空间。

细节01　定义结构体对结构体成员的类型限制

定义一个结构体时，对成员的类型没有限制，可以是任何的基本的数据类型，也可以是数组、联合体或者枚举类型的变量，甚至是结构体变量或者指向结构体变量的指针。

细节02　成员运算符（.）和（->）的差异

成员运算符（.）和指向结构体成员的运算符（->）两者都是用来引用结构体变量成员的，但是他们的应用环境是完全不同的。

成员运算符（.）用在一般结构体变量中，表示结构体变量的成员。

指向结构体成员运算符（->）与指向结构体变量的指针连用，表示指向结构体变量指针的成员。

例如：

```
struct student
{
    int num;
    char name[10];
};
struct student stu, *s=&stu;
```

其中，stu.mum、s->num是正确的引用方法，但是stu->num和s.num是不允许的。其实，s->num就相当于（*s）.num。

细节03　结构体和联合体的区别和联系

结构体和联合体都是C语言的两个组合数据类型，在定义形式以及应用方式上都表现出了很

多的相似之处，但是也有一些不同之处。

联合体是一种特殊形式的结构体，其所有成员共享一个存储空间，存储空间中保存最近更新的成员值。联合体的变量的长度是联合体中各成员中最长的长度。这里所谓的共享一段内存空间，不是把所有的成员同时装进一个联合体变量内，而是指该联合体变量可被赋予任意一个成员值，但是每次只能赋一种值，赋了新值，则替换原有值。

在结构体中各成员都有各自的存储空间，一个结构体变量的总长度是各个成员变量的长度之和。

结构体类型是由不同数据类型（当然也可以是相同的数据类型）的域所构成的，各个域一般都是用来表示不同对象的各个方面的属性，他们在存储区中都占有各自不同的存储空间，任何时刻某个域都不会影响到其他的域。

联合体虽然也包含若干个域，但是各个域都共享同一个存储区，即各个域的存储地址都是相同的。对联合体某个域的修改会直接影响到其他域，并且最近一次的修改对任意成员的赋值都是有效的。

细节 04 enum 的好处

枚举类型是 C 语言中的一种高级数据类型，枚举类型可以用来定义常量数值。声明枚举类型的一般形式如下：

```
enum 枚举类型名
{
    取值表
};
```

在取值表中应列出所有可用值。这些值也称为枚举元素。

注意：取值表中的值称为枚举元素，其含义由程序解释，不是因为写成“sun”就自动代表“星期天”。

例如：

```
enum weekday
{
    sun,mon,tue,wed,thu,fri,sat
};
```

该枚举类型名为 weekday，枚举值共有 7 个，分别是 sun、mon、tue、wed、thu、fri、sat。当一个变量被说明为 weekday 类型时该变量的取值只能是这 7 个枚举值中的一个。

用 enum 关键字说明枚举常量有三点好处。

- 用 enum 关键字说明的常量由编译程序自动生成，程序员不需要用手工对常量进行逐个赋值。

- 在定义 enum 常量的同时也定义了一个枚举类型标识符。
- 在调试程序时通常可以检查枚举常量，这一点是非常有用的，尤其在不得不手工检查头文件中的常量值时。

例如，在上面的代码中，与用#define 说明常量相比，用 enum 说明常量还有其他好处，与用#define 指令说明常量(即说明标识符常量)相比，用 enum 关键字说明常量(即说明枚举常量)有以下几点好处：

使程序更容易维护，因为枚举常量是由编译程序自动生成的，而标识符常量必须由程序员手工赋值。

例如：

```
enum weekday
{
    sun,mon,tue,wed,thu
};
```

在上例中，sun 等枚举常量依次被编译程序自动赋值为 0，1，2 和 3。同样，也可以用#define 指令说明类似的一组常量，请看下例：

```
#define sun 0
#define mon 1
#define tue 2
#define wed 3
#define thu 4
```

上述两例的结果是相同的。假设要增加两个新的常量，例如 fri 和 sat，如果常量原来是用 enum 关键字说明的，可以在原来的常量中的任意一个位置插入这两个常量，因为编译程序会自动赋给每一个枚举常量一个唯一的值；如果常量原来是用#define 指令说明的，就不得不手工为新的常量赋值。在上面的例子中，并不关心常量的实际值，而只关心常量的值是否唯一，因此，用 enum 关键字说明常量使程序更容易维护，并且能防止给不同的常量赋予相同的值。

使程序更易读，这样别人修改你的程序时就比较方便。

使程序调试起来更方便，因为某些标识符调试程序能打印枚举常量的值。这一点在调试程序时是非常用的，因为如果你的程序在使用枚举常量的一行语句中停住了，你就能马上检查出这个常量的值。反之，绝大多数调试程序无法打印标识符常量的值，因此你不得不在头文件中手工检查该常量的值。

细节 05　.h 文件

在编写多个源文件时，什么内容应放到.h 文件中？

一般情况下将如下内容放到.h 文件中。

- 宏定义。

- 结构、联合和枚举声明。
- typedef 声明。
- 外部函数声明。
- 全局变量声明。

细节 06 宏定义问题

宏定义是预处理命令的一种，以#define 开头。它提供了一种可以替换原代码中字符串的机制。根据宏定义中是否有参数，可以将宏定义分为不带参数的宏定义和带参数的宏定义两种。

有时在使用一些定义函数式的宏的时候会出现一些问题，但有的时候又没有问题，这就在定义时注意一些问题。

宏扩展必须使用括号，来保护表达式中低优先级的操作符，以便确保调用时达到想要的效果。

例如：

```
#define ADD ( n ) n+n
```

调用：

```
5*ADD ( n )
```

则会被扩展为：

```
5*n+n
```

而本意是希望得到：

```
5* ( n+n )
```

解决的办法就是上面说的宏扩展时加上括号就能避免这种错误发生。

在宏定义内部，所有参数的出现都必须用括号括起来，以便保护实参中任何低优先级的操作符不受宏扩展其他部分的影响。

例如：

```
#define MUL ( n ) n*n
```

调用：

```
MUL ( n+1 )
```

则会被扩展为：

```
n+1*n+1
```

而本意是希望得到：

```
(n+1)*(n+1)
```

如果一个参数在扩展中出现多次且实参是带其他作用的表达式，则宏将不能正确的运行。

例如：

```
#define MUL ( n )  n*n
```

调用：

```
MUL ( i++ )
```

则会被扩展为：

```
i++*i++
```

在同一个表达式中使用导致同一对象修改两次或修改以后又被检查的自增或自减或赋值操作符时，则认为是未定义的。

细节 07　条件编译

当两段代码共存于一个文件，编译时有选择地编译其中的一部分，这时该如何实现呢？

可以用以下两种的办法实现：

在源码中使用条件编译语句，然后在程序文件中定义宏的形式来选择需要的编译代码。

在源码中使用条件编译语句，然后在编译命令的命令中加入宏定义命令来实现选择编译。

细节 08　文件打开

文件指针是一个指向文件有关信息的指针，这些信息包括文件名、状态和当前位置，他们保存在一个结构体变量中。该结构体类型是由系统定义的，C 语言规定该类型为 FILE 型，其声明如下：

```
typedef struct
{
    short level;
    unsigned flags;
    char fd;
    unsigned char hold;
    short bsize;
    unsigned char *buffer;
    unsigned ar *curp;
    unsigned istemp;
    short token;
}FILE;
```

编写程序时可用上面定义的 FILE 类型来定义变量，注意在定义变量时不用将结构体内容全部给出，只需写成如下形式：

```
FILE *fp;
```

说明 fp 是一个指向 FILE 类型的指针变量。

fopen 函数用来打开一个文件，其调用的一般形式为：

```
FILE *fp;
fp=fopen(文件名，使用文件方式);
```

其中“文件名”是将要被打开文件的文件名；“使用文件方式”是指对打开的文件是要进行读还是写。使用文件方式如表 8.1 所示。

表 8.1　　使用文件方式

文件使用方式	含　义
“r”（只读）	打开一个文本文件，只允许读数据
“w”（只写）	打开或建立一个文本文件，只允许写数据
“a”（追加）	打开一个文本文件，并在文件末尾写数据
“rb”（只读）	打开一个二进制文件，只允许读数据
“wb”（只写）	打开或建立一个二进制文件，只允许写数据
“ab”（追加）	打开一个二进制文件，并在文件末尾写数据
“r+”（读写）	打开一个文本文件，允许读和写
“w+”（读写）	打开或建立一个文本文件，允许读写
“a+”（读写）	打开一个文本文件，允许读，或在文件末追加数据
“rb+”（读写）	打开一个二进制文件，允许读和写
“wb+”（读写）	打开或建立一个二进制文件，允许读和写
“ab+”（读写）	打开一个二进制文件，允许读，或在文件末追加数据

如果要以只读方式打开文件名为 happy 的文本文档文件，应写成如下形式：

```
FILE *fp;
fp=("happy.txt","r");
```

对表 8.1 中给出的使用文件方式有以下几个方面强调一下。

（1）当用“r”打开一个文件时，该文件必须是已经存在的，只能读取该文件中的内容，不能向该文件中写入任何内容。

（2）当用“w”打开文件时会向指定的文件中写入内容。若打开的文件不存在，则会新建一个文件，新建的文件以打开的文件名为文件名，若打开的文件已经存在，则会用新建的文件覆盖原有的文件，并向新文件中写入内容。

（3）若要向一个已存在的文件末尾追加新的数据，只能用“a”方式打开文件。但此时该文件必须是存在的，否则将会出错。

（4）在打开一个文件时，如果出错，fopen 函数会带回一个出错信息。此时 fopen 函数将带回

一个空指针值 NULL。

看下面这段代码：

```
FILE *fp;
fp=fopen(filename,'r');
```

实际上这两句代码并不能正确的打开 filename 所指定的文件，是因为程序中写的是'r'，而不是"r"，也就是说 fopen 函数中 mode 参数必须是字符串而不是字符，这点经常会被一些人忽视，要引起大家重视。

细节 09　C 语言中的预处理功能

预处理是 C 语言的一个重要功能，它由预处理程序负责完成。所谓预处理是指在进行编译的第一遍扫描（词法分析和语法分析）之前所做的工作。对一个源文件进行编译时，如果程序中有预处理命令，系统将自动引用预处理程序对源程序中预处理部分所做的处理，处理完毕自动进入对正常程序的编译。

在 C 语言中包含以下几方面的预处理功能。

（1）宏定义：包含无参数宏定义和有参数定义。

（2）文件包含：是指一个源文件可以把另一个源文件完全包含进来。

（3）条件编译：对源程序中的程序段进行有选择的编译。

预处理命令语句和一般 C 语言的语句不同。所有的预处理语句都是以#开头，单独占一行，不用分号结尾，并且一般放在程序的开始处。

细节 10　使用宏时需要注意些什么

在使用宏时应该注意以下几点。

（1）宏定义不是 C 语言的语句，不要在行末添加分号。

（2）书写宏定义的时候，如果需要续行，有些编译器需要加续行符（\）。

（3）在宏定义时，宏名和带参数的括号间不要加空格，否则该空格将被看做是宏体的一部分。

（4）进行宏替换时，不应增加或者减少东西，只是简单的替换。

细节 11　文件包含的嵌套

包含文件也是可以进行嵌套的，一个被包含的文件 file1.c，又包含了另一个文件 file2.c，就是文件的嵌套。

下面介绍一下文件嵌套的形式。

在文件 file1.c 中：

```
#include "file2.c"
```

表示一个被包含的文件 file1.c 又包含了另一个文件 file2.c，在文件 file2.c 中：

```
#include "file3.c"
```

这就是说文件 file1.c 包含文件 file2.c，在文件 file2.c 中包含文件 file3.c，也就是说 file1.c 包含了两个文件，file2.c 和 file3.c。

如果在文件 file2.c 中没有包含文件 file2.c 的内容，也可以在文件 file1.c 中进行包含，在 file1.c 中就用到两个 include 命令：

```
#include "file3.c"
#include "file2.c"
```

这里要注意命令的顺序，file3.c 要写在 file2.c 的前面。

细节 12 malloc()与 calloc()的区别

1. malloc()函数

mallo 函数用于分配内存。其函数的原型如下：

```
void *malloc( unsigned int size );
```

在 stdlib.h 头文件中包含该函数，作用是在内存中动态的分配一块 size 大小的内存空间。malloc 函数会返回一个指针，该指针指向分配的内存空间，如果出现错误返回 NULL。

注意： 需要注意的是，使用 malloc 函数分配的内存空间是在堆中，而不是在栈中。所以在使用完这块内存之后一定要将其释放掉，释放内存空间使用的函数是 free 函数（下面将会进行介绍）。

2. calloc()函数

calloc 函数包含在头文件 stdlib.h 中，该函数的功能是在内存中动态分配 n 个长度为 size 的连续内存空间数组。calloc 函数会返回一个指针，该指针指向动态分配的连续内存空间地址。当分配空间错误时，返回 NULL。

其函数的原型如下：

```
void * calloc(unsigned n, unsigned size);
```

例如，使用该函数分配一个整型数组内存空间：

```
int* pArray;                                    /*定义指针*/
pArray=(int*)calloc(3,sizeof(int));             /*分配内存数组*/
```

在上面的代码中 pArray 为一个整型指针，使用 calloc 分配内存数组，第一个参数表示分配数组中元素的个数，而第二个参数表示元素的类型。最后将返回的指针赋给 pArray 指针变量，pArray 指向的就是该数组的首地址。

3. malloc 和 calloc 函数的区别

参数上的区别：

```
malloc(size_t size);
calloc(size_t n,size_t size);
```

malloc 分配一块 size 大小的内存块，而 calloc 分配一个 n*size 大小的内存块。

返回内存块的状态不同。

malloc 分配的内存块没有被清零，而 calloc 分配的内存块是清了零的。

细节 13　什么是“内存泄露”

当定义了一个指针时，立即要为这个指针分配一个内存空间，这只是防止了野指针的产生，当使用完毕以后应立即释放其所占空间。

什么是“野指针”？

“野指针”是指指向“垃圾”内存的指针，而不是 NULL 指针。“野指针”形成的原因主要有。

（1）指针变量没有初始化。

（2）指针被释放或者删除以后，没有设置为 NULL，被误认为是合法的指针

（3）指针的操作超出了变量的作用范围。这种情况非常不容易发现。

如果没有及时释放指针所占用的内存空间，而下次再使用这个指针时又为其分配内存空间，再使用，再分配。这样次数一多，内存空间就被慢慢消耗掉了，所以，将这种现象称为内存泄露。

细节 14　内存耗尽了怎么办

如果在申请动态内存时找不到足够大的内存块，malloc()函数和 new()函数将返回 NULL 指针，宣告内存申请失败。

解决内存耗尽问题可以使用以下三种方法。

（1）判断指针是否为 NULL，如果是，则马上使用 return 语句终止本函数。

（2）判断指针是否为 NULL，如果是，则用 exit(1)终止整个程序的运行。

（3）为 new()函数和 malloc()函数设置异常处理函数。

细节 15 常见的内存管理错误有哪些

C 语言只对程序进行语法检查而对语意并不进行检查，需要程序员自行控制。这样就极易出现内存管理错误，而这类错误是不宜发现的。

C 语言中常见的内存管理错误有以下 4 种。

（1）分配了一块内存空间，引用了未经初始化的内容。

（2）释放了一块内存空间，但是却继续引用它的内容。

（3）分配了一块内存空间，但是没有保存指向该内存的指针，造成无法对此指针进行引用或者释放，这块内存空间也成了系统中“垃圾“内存，从而浪费了内存资源。

（4）内存的读写操作超越了所分配的内存空间的边界。

细节 16 标准文件和一般文件有什么区别

C 语言的文件按其操作方法可分为标准文件和一般文件两类，这两者有何区别?

标准文件是指系统为自动分配缓冲区的文件，这种文件只有 3 个，分别是标准输入文件、标准输出文件和标准出错信息文件。

标准输入文件是指键盘输入，标准输出文件是指显示屏幕输出，标准出错信息文件是指将出错信息显示在屏幕上。

标准文件由系统自动打开，自动分配文件指针，分别是 stdin，stdout 和 stderr。当退出系统时，自动关闭标准文件。

一般文件是指除了标准文件之外的其他磁盘文件和设备文件，在使用前要先用打开文件函数将文件打开，如果正常打开可得到一个文件指针，可利用该文件指针对文件进行操作。不用时可以使用文件关闭函数关闭文件。

打开和关闭文件的操作是标准文件和一般文件的主要区别。另外，对标准文件的读写函数和一般文件的读写函数也是不相同的。

细节 17 C 语言的常用库文件包含哪些

C 语言提供了丰富的系统文件，称为库文件，C 的库文件分成两类，一类是扩展名为.h 的文件，

称为头文件，在头文件中包含了常量定义、类型定义、宏定义、函数原型以及各种编译选择设置等信息。另一类是函数库，包括了各种函数的目标代码，供用户在程序中调用。

库函数在使用前需要包含该函数原型坐在的.h 文件。常用的库文件如表 8.2 所示。

表 8.2　　常用的库文件

库 文 件 名	功 能 说 明
alloc.h	内存管理函数（分配、释放等）
assert.h	定义 assert 调试宏
bios.h	调用 IBM-PC ROM BIOS 子程序的各个函数
conio.h	调用 DOS 控制台 I/O 的函数
ctype.h	包含有关字符分类及转换的各类信息（如 isalpha 和 toascii 等）
dir.h	包含有关目录和路径的结构、宏定义和函数
dos.h	定义和说明 MSDOS 和 8086 调用的一些常量和函数
erron.h	定义错误代码的助记符
pcntl.h	定义在打开库子程序连接时的符号常量
float.h	包含有关浮点运算的一些参数和函数
graphics.h	有关图形功能的各个函数，图形错误代码的常量定义，针对不同驱动程序的各种颜色值以及函数用到的一些特殊结构
io.h	包含低级 I/O 子程序的结构和说明
limit.h	包含环境参数。编译时间限制、数的范围等信息
math.h	数学运算函数
mem.h	说明一些内存操作函数（其中大多数也在 string.h 中说明）
process.h	进程管理中的各个函数，spawn()和 exec()函数的结构说明
setjmp.h	定义 longjmp()和 setjmp()函数用到的 jmp_buf 类型说明这两个函数
share.h	定义文件共享函数的参数
signal.h	定义了信号原子类型、信号处理函数的注册、发送信号的函数等
stdarg.h	定义读函数参数表的宏
stddef.h	定义一些公共数据类型和宏
stdio.h	定义标准和扩展的类型的宏，还定义了标准 I/O 预定义流，stdin，stdout 和 stderr，说明 I/O 流子程序
stdlib.h	说明一些常用的子程序。如转换子程序、搜索/排序子程序等
string.h	串操作和内存操作函数
sys\types.h	定义 ftime()函数和 timeb 结构
sys\stat.h	定义在打开和创建文件时用的一些符号常量
time.h	定义时间转换子程序 asctime、localtime 和 gmtime 的结构，ctime、difftime、gmtime、localtime 和 stime 用到的类型，并提供这些函数的原型
value.h	定义一些重要的常量，包括依赖于计算机硬件的和为与 Unix System V 相兼容而说明的一些常量，包括浮点和双精度值的范围

细节 18 为什么文件打开以后要及时关闭

文件在未打开前是存放在磁盘等外设上的，在文件使用前需要先将其从磁盘读入到主存，文件的打开就是完成了这个目的，即将制定文件的相关信息读取到主存中，以方便访问。

文件的关闭与文件的打开是相对应的操作，为了加快与文件数据交换的效率，在文件使用完毕以后就必须要关闭。文件打开时，系统即为文件开辟了一个缓冲区，每次读写数据时都写在相应的缓冲区中，并没有写在磁盘中，在文件关闭前要将缓冲区中的文件写入到磁盘中，保证所做的修改不会丢失。

另外，文件打开以后及时关闭还有两个好处。第一，能够保证打开文件的安全。如果文件被打开不关闭，数据在缓冲区中是不安全的，很容易对已经打开的文件造成误操作。第二，文件打开会占用一定的内存空间，如果文件打开以后都不关闭，会使可用的内存空间变少，影响计算机的运行效率。

4

第 4 篇
编程技巧
——程序员的神兵利器

第 9 章　C 语言开发人员必知必会的基本技能开发技巧

第 10 章　指针开发技巧

第 11 章　文件开发技巧

第 12 章　数学应用开发技巧

第 13 章　C 语言开发人员必知必会的数据结构开发应用技巧

第 14 章　C 语言开发人员必知必会的高级开发技巧



第9章 C语言开发人员必知必会的基本技能开发技巧

在使用 C 语言开发过程中，经常需要用到各种技巧，如将十进制数转换为十六进制、N 进制转换为十进制、IP 地址形式输出、判断闰年、序列求和等。通过学习本章，程序开发人员可以掌握这些基本技能，以便在以后的开发中直接使用。

技巧 01 十进制数转换为十六进制

在日常生活中，一般主要使用十进制数，例如 10 个 1 元等于 10 元，10 个 10 元等于 100 元。

而在编写 C 程序中，时常为了提高效率或其他一些原因，还要使用八进制或十六进制数，这就需要将十进制数进行转换。十进制和十六进制数之间可以直接转换，利用格式输入函数 scanf()，从键盘中获得一个十进制整数，利用格式输出函数将转换后的结果输出，并不需要复杂的过程。

例 9.1 通过利用输入、输出格式控制符进行转换。在 C 语言程序中，“%x 和%d”分别代表十六进制和十进制数的输入和输出格式控制符（光盘位置：光盘\MR\Instance\9\01）。

```
#include <stdio.h>
main()                                          /*main()主函数*/
{
    int i;                                      /*定义一个变量 i*/
    printf("please input decimalism number:\n");  /*双引号内普通字符原样输出并换行*/
    scanf("%d", &i);                            /*scanf()函数以十进制形式获得 i 的值*/
    printf("the hex number is %x\n",i);         /*将 i 的值以十六进制形式输出*/
}
```

运行本实例，效果如图 9.1 所示。

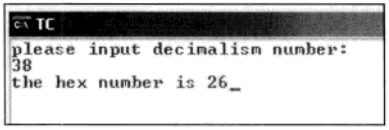


图 9.1 十进制转换为十六进制的效果

注意： 在使用 printf()及 scanf()函数时要注意格式，尤其是 scanf()函数，scanf(“%d”,&i);中的&符号不要忘记写。

技巧 02　N 进制转换为十进制

为了进制间灵活的转换，编程实现任意输入一个数并输入几进制，则将其转换后的十进制数输出。

实现步骤如下。

程序中用到了字符串函数 strupr()和 strlen()函数，前者是将括号内指定字符串中的小写字母转换为大写字母，其余字符串不变，后者是求括号中指定字符串的长度，即有效字符的个数，使用这两个函数应在程序开头写上一行：

```
#include<string.h>
```

本题的主要思路是用字符型数组 a 存放一个 n 进制数，再对数组中的每个元素进行判断，如果是 0～9 的数字则进行以下处理：

```
t = a[i] - '0';
```

如果是字母则进行以下处理：

```
t = a[i] - 'A' + 10;
```

如果输入的数据与进制不符，则输出数据错误并退出程序。

例 9.2　从键盘中输入数据并输入其进制，如果数据与进制不符则输出错误并退出程序，否则判断是数据还是字母并进行相应处理(光盘位置：光盘\MR\Instance\9\02)。

```
#include <stdio.h>                                          /*引用头文件*/
#include <string.h>
main()                                                      /*main()函数*/
{
    long t1;                                                /*定义变量*/
    int i, n, t, t3;                                        /*定义整型变量*/
    char a[100];                                            /*定义字符型数组*/
    printf("please input a number string:\n");              /*输出提示信息*/
    gets(a);                                                /*输入 n 进制数存到数组 a 中*/
    strupr(a);                                              /*将 a 中的小写字母转换成大写字母*/
    t3 = strlen(a);                                         /*求出数组 a 的长度*/
    t1 = 0;                                                 /*为 t1 赋初值 0*/
    printf("please input n(2or8or16):\n");                  /*输出提示信息*/
    scanf("%d", &n);                                        /*输入进制数*/
    for (i = 0; i < t3; i++)                                /*for 循环语句*/
    {
        if (a[i] - '0' >= n && a[i] < 'A' || a[i] - 'A' + 10 >= n)   /*判断输入的数据和进制数是否相符*/
        {
            printf("data error!!");                         /*输出错误*/
            exit(0);                                        /*退出程序*/
        }
        if (a[i] >= '0' && a[i] <= '9')                     /*判断是否为数字*/
            t = a[i] - '0';                                 /*求出该数字赋给 t*/
        else if (n >= 11 && (a[i] >= 'A' && a[i] <= 'A' + n - 10))   /*判断是否为字母*/
            t = a[i] - 'A' + 10;                            /*求出字母所代表的十进制数*/
```

```
        t1 = t1 * n + t;                                    /*求出最终转换成的十进制数*/
    }
    printf("the decimal is %ld\n", t1);                     /*将最终结果输出*/
}
```

运行本实例，效果如图 9.2 所示。

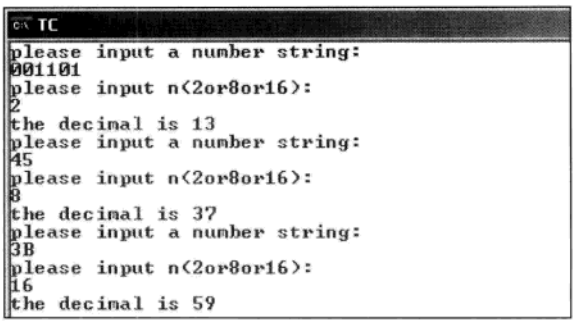


图 9.2　N 进制转换为十进制的效果

技巧 03　IP 地址形式输出

根据特殊原因需要，常常会将任意输入 32 位的二进制数，编程实现将该二进制数转换成 IP 地址形式。例如：

输入：11111111111111111111111100000000

输出：255.255.255.0

例 9.3　将输入的二进制数以每 8 位数为一个单位分开，再通过自定义的函数将这 8 位二进制数转换成对应的十进制数即可（光盘位置：光盘\MR\Instance\9\03）。

```
#include<stdio.h>                                           /*引用头文件*/
int bin_dec(int x, int n)                                   /*自定义函数将二进制数转换成十进制数*/
{
    if (n == 0)                                             /*判断 n 的值*/
    /*递归结束条件*/
    {
        return 1;
    }
    return x *bin_dec(x, n - 1);                            /*递归调用 bin_dec()函数*/
}
main()                                                      /*mian()主函数*/
{
    int i;                                                  /*定义变量*/
    int ip[4] ={0};
    char a[33];                                             /*存放输入的二进制数*/
    printf("please input binary number:\n");
    scanf("%s", a);                                         /*二进制数以字符串形式读入*/
    for (i = 0; i < 8; i++)                                 /*for 循环语句，i 取值范围为 0~7*/
    {
```

```
        if (a[i] == '1')                                    /*判断数组中的值是否为 1*/
        {
            ip[0] += bin_dec(2, 7-i);                       /*计算 0-7 转换的结果*/
        }
    }
    for (i = 8; i < 16; i++)                                /*i 的取值范围为 8~16*/
    {
        if (a[i] == '1')                                    /*判断数组中的值是否为 1*/
        {
            ip[1] += bin_dec(2, 15-i);                      /*计算 8-15 转换的结果*/
        }
    }
    for (i = 16; i < 24; i++)                               /*i 的取值范围为 16~23*/
    {
        if (a[i] == '1')                                    /*判断数组中的值是否为 1*/
        {
            ip[2] += bin_dec(2, 23-i);                      /*计算 16-23 转换的结果*/
        }
    }
    for (i = 24; i < 32; i++)                               /*i 的取值范围为 24~23*/
    {
        if (a[i] == '1')                                    /*判断数组中的值是否为 1*/
        {
            ip[3] += bin_dec(2, 31-i);                      /*计算 24-31 转换的结果*/
        }
        if (a[i] == '\0')                                   /*判断数组中的值是否为 0*/
        {
            break;                                          /*跳出本次循环*/
        }
    }
    printf("ip:\n");
    printf("%d.%d.%d.%d\n", ip[0], ip[1], ip[2], ip[3]);    /*将最终结果以 ip 形式输出*/
}
```

运行本实例，效果如图 9.3 所示。

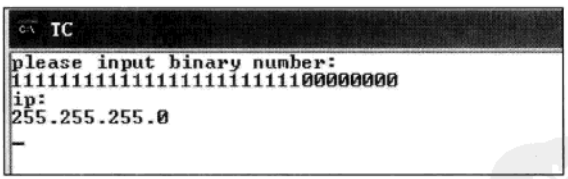


图 9.3　格式化日期后的效果

技巧 04　数字由小到大排序

任意输入 3 个整数，编程实现对这 3 个整数由小到大排序，并将排序后的结果显示在屏幕上。

实现技巧：

本技巧中用到了 if 语句，if 语句的 3 种形式如下：

- 第一种形式是单条件单分支的 if 语句：

```
if(表达式)
   语句
```

其语义是：如果表达式的值为真，则执行其后的语句,否则不执行该语句。

- 第二种形式是单条件双分支的 if 语句：

```
if(表达式)
   语句 1
else
   语句 2
```

其语义是：如果表达式的值为真，则执行语句 1，否则执行语句 2。

- 第 3 种形式是多条件多分支的 if 语句：

```
if(表达式 1)
   语句 1
else  if(表达式 2)
   语句 2
else  if(表达式 3)
   语句 3
…
else  if(表达式 m)
   语句 m
else  语句 n
```

其语义是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整个 if 语句之外继续执行程序；如果所有的表达式均为假，则执行语句 n 。 然后继续执行后续程序。

3 种形式的 if 语句中在 if 后面都有“表达式”，这个“表达式”一般为逻辑表达式或关系表达式。在执行 if 语句时先对表达式求解，若表达式的值为 0，按“假”处理，若表达式的值为非“0”，按“真”处理，执行指定的语句。

else 子句不能作为语句单独使用，它必须是 if 语句的一部分，与 if 配对使用。

if 与 else 后面可以包含一个或多个内嵌的操作语句，当为多个操作语句时要用“{}”将几个语句括起来成为一个复合语句。if 语句可以嵌套使用，即在 if 语句中又包含一个或多个 if 语句，在使用时应注意 else 总是与它上面最近的未配对的 if 配对。

例 9.4 使用 if 语句进行条件判断，如果 a 大于 b，则借助中间变量 t 实现两个变量 a 与 b 值的互换，依此类推进行 a 与 c 的比较，b 与 c 的比较，从而最终得到的结果是 a,b,c 依次由小变大排列（光盘位置：光盘\MR\Instance\9\04）。

```
#include <stdio.h>                                  /*引用头文件*/
main()
{
    int a, b, c, t;                                 /*定义 4 个基本整型变量 a, b, c, t*/
    clrscr();                                       /*清屏*/
    printf("please input a,b,c:\n");                /*双引号内普通字符原样输出并换行*/
    scanf("%d%d%d", &a, &b, &c);                    /*输入任意 3 个数*/
    if (a > b)                                      /*如果 a 大于 b，借助中间变量 t 实现 a，b 值互换*/
    {
```

```
        t=a;                                    /*将 a 的值赋给 t*/
        a=b;                                    /*将 b 的值赋给 b*/
        b=t;                                    /*将 t 的值赋给 b*/
    }
    if (a > c)                                  /*如果 a 大于 c，借助中间变量 t 实现 a、c 值互换*/
    {
        t=a;                                    /*将 a 的值赋给 t*/
        a=c;                                    /*将 c 的值赋给 c*/
        c=t;                                    /*将 t 的值赋给 c*/
    }
    if (b > c)                                  /*如果 b 大于 c，借助中间变量 t 实现 b、c 值互换*/
    {
        t = b;                                  /*将 b 的值赋给 t*/
        b = c;                                  /*将 c 的值赋给 b*/
        c = t;                                  /*将 t 的值赋给 c*/
    }
    printf("the order of the number is:\n");    /*显示提示信息*/
    printf("%d,%d,%d", a, b, c);                /*输出函数将 a、b、c 的值顺序输出*/
}
```

运行本实例，效果如图 9.4 所示。

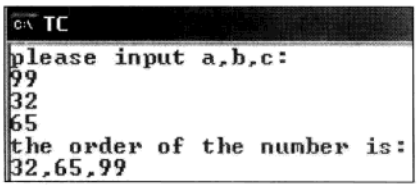


图 9.4　3 个数由大到小排序

注意： 本实例中使用 scanf("%d%d%d",&a,&b,&c)从键盘中获得任意 3 个数，在输入数据时在两个数据之间以一个或多个空格间隔，也可以使用回车键、跳格键<Tab>，一定不能用逗号作为两个数据间的分隔符。当使用 scanf("%d,%d,%d",&a,&b,&c);格式输入函数时，在输入数据时两个数据之间要用“,”做间隔。

技巧 05　阶梯问题

在你面前有一条长长的阶梯。如果你每步跨 2 阶，那么最后剩 1 阶；如果你每步跨 3 阶，那么最后剩 2 阶；如果你每步跨 5 阶，那么最后剩 4 阶；如果你每步跨 6 阶，那么最后剩 5 阶；只有当你每步跨 7 阶时，最后才正好走完，一阶也不剩。请问这条阶梯至少有多少阶？（求所有三位阶梯数）

实现思路：

本实例的关键是如何来写 if 语句中的条件，如果这个条件大家能够顺利地写出，那整个程序也基本上完成了。条件如何来写这主要是根据题意，出题的规律就是总的阶梯数对每步跨的阶梯数取余得的结果就是剩余阶梯数，这 5 种情况是&&的关系，也就说必须同时满足。

因为题目要求三位阶梯数，所以求出的数值一定是在大于等于 100 小于 1000 范围内，所以使用 for 循环对大于等于 100 小于 1000 的所有三位整数进行筛选。使用 if 语句根据题意设置相应的条件，如果满足条件则输出该结果 ，否则程序继续下次循环。

例 9.5 使用 for 语句中嵌套选择判断条件 if 语句的方式，解决阶梯问题。（光盘位置：光盘\MR\Instance\9\05）

```
#include <stdio.h>                                          /*引用头文件*/
main()
{
    int i;                                                  /*定义基本整型变量 i*/
    /*for 循环求 100 到 1000 的所有 3 位数*/
    for (i = 100; i < 1000; i++)
        /*根据题意写出对应的条件*/
        if (i % 2 == 1 && i % 3 == 2 && i % 5 == 4 && i % 6 == 5 && i % 7 == 0)
            printf("the number of the stairs is %d\n", i);  /*输出阶梯数*/
}
```

运行本实例，效果如图 9.5 所示。

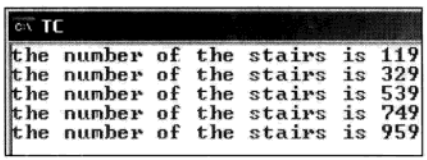


图 9.5 使用 Stack 类实现递归功能

技巧 06 判断闰年

一年是 365 天，但每四年中会有一年是闰年，闰年一年是 366 天，怎么判断某年是不是闰年呢？本实例就来介绍相关问题。

实现思路：

如果某年能被 4 整除但不能被 100 整除，或者该年能被 400 整除则该年为闰年。按照上述思想编写的公式为：

```
year%4==0&&year%100!=0)||year%400==0
```

除本解决方法外，判断闰年还有许多方法，下面给出的算法（伪代码描述）也为其中一种：

```
{
    if ( 某年能被 400 整除 )
    输出是闰年;
    else  if ( 该年能被 100 整除 )
    输出不是闰年;
    else  if ( 该年能被 4 整除 )
    输出是闰年;
    else
    输出不是闰年;
}
```

将判断闰年的自然语言转换成 C 语言要求的语法形式时需要用到逻辑运算符&&、||、!，这些逻辑符号具体使用规则如下。

- &&逻辑与（相当于其他语言中的 AND），a&&b，若 a，b 为真，则 a&&b 为真。
- ||逻辑或（相当于其他语言中的 OR），a||b，若 a、b 之一为真，则 a||b 为真。
- ！逻辑非（相当于其他语言中的 NOT），a 若为真，则！a 为假。

说明：三者的优先次序是:！→&&→||，即“!”为三者中优先级最高的。

例 9.6　从键盘上输入一个表示年份的整数，判断该年份是否是闰年，判断后的结果显示在屏幕上。（光盘位置：光盘\MR\Instance\9\06）

```
#include <stdio.h>                                              /*引用头文件*/
main()
{
    int year;                                                   /*定义基本整型变量 year*/
    printf("please input the year:\n");
    scanf("%d", &year);                                         /*从键盘输入表示年份的整数
*/
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)  /*判断闰年条件*/
        printf("%d is a leap year", year);                      /*满足条件的输出是闰年*/
    else
        printf("%d is not a leap year", year);                  /    *否则输出不是闰年*/
}
```

运行本实例，效果如图 9.6 所示。

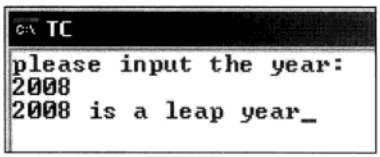


图 9.6　判断闰年

技巧 07　对调数问题

一个两位正整数，将它的个位和十位对调，得到另一个两位正整数，我们称这两个正整数互为对调数。现给定一个两位正整数，问能否找出另一个对调后不同的两位正整数，使得这两个正整数之和等于它们各自的对调数之和？例如：63+47=36+74。编写程序为输入的一个两位正整数找到满足如上要求的数或输出“不存在”。

实现步骤：

（1）定义数据类型及判断输入的数是否为符合要求的两位正整数。如不符合条件则提示重新输入，符合条件则通过运算求出其对调数。

（2）寻找另一个满足条件的两位数，若该数满足条件则继续判断其是否符合两数相加等于这两个数的对调数相加之和，且这两个数不互为对调数的条件，若满足该条件则按题中要求格式输出，不满足条件结束本次循环继续下次循环条件的判断。

（3）定义一个变量用于在程序中计数，若此变量等于 0，这说明没有满足上面条件的两位数，则输出不存在。

例 9.7 使用循环语句和条件判断语句解决对调数问题。（光盘位置：光盘\MR\Instance\9\07）

```
#include<stdio.h>                                   /*引用头文件*/
main()
{
  int x,y,z,x1,y1,z1,i,k,n,j=0;                     /*定义变量的数据类型*/
  while(1)                                          /*while 为真*/
  {
    printf("please input an integer\n");            /*输出提示*/
    scanf("%d",&n);                                 /*输入函数获得 n 的值*/
    if(n<=10||n>=100)                               /*当输入的数小于等于 10 或大于等于 100 时输出错误*/
    {
      printf("data error\n");
      continue;                                     /*跳出本次循环*/
    }
    else if(n%10==0)                                /*当这个两位数对 10 取余等于 0 时输出错误*/
    {
    printf("data error\n");
    continue;                                       /*跳出本次循环*/
    }
    else
    {
      x=n/10;                                       /*x 为输入的两位数中的十位*/
      y=n%10;                                       /*y 为输入的两位数中的个位*/
      z=10*y+x;                                     /*z 为 n 的对调数*/
      break;                                        /*跳出循环*/
    }
  }
  for(i=11;i<100;i++)                               /*i 遍历所有的两位数*/
  {
    if(i%10==0)                                     /*除去所有的对调数是个位的数*/
    continue;                                       /*结束本次循环*/
    else
    {
      x1=i/10;                                      /*x1 为 i 的十位*/
      y1=i%10;                                      /*y1 为 i 的个位*/
      z1=10*y1+x1;                                  /*z1 为 i 的对调数*/
      /*当满足 n 加 i 等于 n 的对调数加 i 的对调数且 n 和 i 不互为对调数*/
      if(n+i==z+z1&&n!=z1)
      {
        printf("%d+%d=%d+%d\n",n,i,z,z1);           /*输出题中要求的表达式形式*/
        j++;                                        /*j 自加*/
      }
      else
      continue;                                     /*否则结束本次循环*/
    }
  }
  if(j==0)                                          /*判断 j 是否等于 0*/
  printf("inexistince");                            /*条件成立输出不存在*/
}
```

运行本实例，效果如图 9.7 所示。

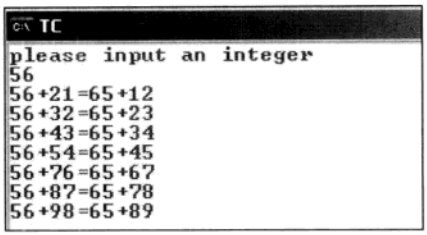


图 9.7　对调数问题

技巧解析：

（1）实例中多次用到 continue 和 break，以下是 continue 和 break 使用的相关要点。

- continue 语句的一般形式：

```
continue;
```

其作用为结束本次循环，即跳过循环体中下面还未执行的语句，接着进行下一次是否执行循环的判断。

- break 语句的一般形式：

```
break;
```

其作用为提前结束循环，即不再执行循环体中后面的语句及其余的几次循环。

- continue 和 break 语句的区别是：

continue 语句只结束本次循环，而不是终止整个循环的执行。而 break 语句则是结束整个循环过程，不再判断执行循环的条件是否成立。

（2）本实例的另一个要点就是如何求一个两位数十位上的数和个位上的数，这里求十位上的数我们采用对 10 整除的方法，求个位上的数采用对 10 取余的方法，通过这两次运算就可很容易求出一个两位整数的对调数。

技巧 08　评定成绩等级

有些学校或有些时候并不是直接给出学生分数，或者有时需要对学生成绩分等级，这时就可以用到本技巧。编程对输入的百分制分数给出相应的等级，当分数大于等于 90 时为“A”，80～89 为“B”，70～79 为“C”，60～69 为“D”，60 分以下为“E”。

实现思路：

输入成绩，当输入的成绩是 100 时，我们可以把它看成 90 分，因为 100 和 90 分都是 A 等的，

对输入的成绩使用除法便求出其高位数字，通过高位数字就可以判断出输出的成绩在哪个等级。使用 switch 语句对取出的高位数字进行选择判断，如果高位数为 9 则是 A 等，如果是 8 则是 B 等，依此类推，分别出现 C、D、E 等。

例 9.8 利用算术运算符“/”取输入成绩的高位数，使用 switch 语句对取出的高位数做选择，根据不同数字，给出不同等级。(光盘位置：光盘\MR\Instance\9\08)

```
#include<stdio.h>
main()
{
    int score;
    printf("\nplease enter score(score<=100):");
    scanf("%d", &score);                                    /*输入学生成绩*/
    if (score == 100)                                       /*如果成绩是 100 则和 90 是一样的等级*/
        score = 90;
    score = score / 10;                                     /*求出成绩的高位数字*/
    switch (score)
    {
        case 9:
            printf("the grade is A");                       /*等级为 A*/
            break;
        case 8:
            printf("the grade is B");                       /*等级为 B*/
            break;
        case 7:
            printf("the grade is C");                       /*等级为 C*/
            break;
        case 6:
            printf("the grade is D");                       /*等级为 D*/
            break;
        default:
            printf("the grade is E");                       /*等级为 E*/
    }
}
```

运行本实例，效果如图 9.8 所示。

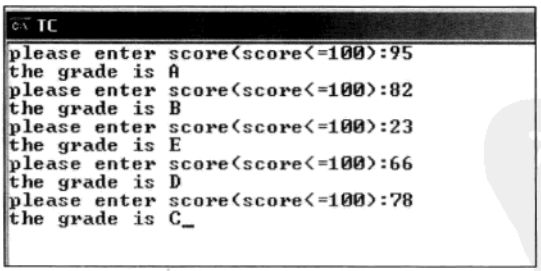


图 9.8 评定成绩等级

技巧 09 打印乘法口诀表

“一一得一，一二得二，一三得三……”相信在上小学的时候每个人都背过乘法口诀，有些地方又叫做“九九乘法表”，本技巧就是将乘法口诀表打印在屏幕上。

实现思路：

打印乘法口诀表关键是要分析程序的算法思想，本实例中两次用到 for 循环，第一次 for 循环看成是乘法口诀表的行数，同时也是每行进行乘法运算的第一个因子，第二个 for 循环范围的确定建立在第一个 for 循环的基础上，即第二个 for 循环的最大取值是第一个 for 循环中变量的值。

例 9.9　使用 for 循环嵌套语句，将乘法口诀表打印出来。(光盘位置：光盘\MR\Instance\9\09)

```
#include<stdio.h>
main()
{
    int i, j;                                          /*定义 i、j 两个变量为基本整型*/
    for (i = 1; i <= 9; i++)                           /*for 循环 i 为乘法口诀表中的行数*/
    {
        /*乘法口诀表中的另一个因子，取值范围受一个因子 i 的影响*/
        for (j = 1; j <= i; j++)
            printf("%d*%d=%d ", i, j, i *j);           /*输出 i、j 及 i*j 的值*/
        printf("\n");                                  /*打完每行值后换行*/
    }
}
```

运行本实例，效果如图 9.9 所示。

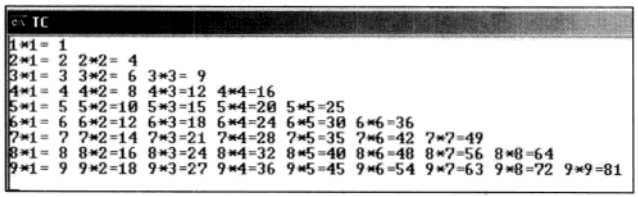


图 9.9　打印乘法口诀表

技巧 10　打印杨辉三角

打印出以下的杨辉三角形（要求打印出 10 行）。

1

1　1

1　2　1

1　3　3　1

1　4　6　4　1

1　5　10 10　5　1

……

技巧 8 是使用 for 循环嵌套的方法实现的，本技巧同样也需要用到 for 循环嵌套才能实现，不

过本技巧更复杂一些，不但要使用 for 嵌套语句，还需要使用二维数组的相关知识才能实现。

实现思路：

要想打印出杨辉三角首先得找出图形中数字间的规律，从图形中我们分析出这些数字间有以下规律：

- 每一行的第一列均为 1；
- 对角线上的数字也均为 1；
- 除每一行第一列和对角线上数字外，其余数字均等于其上一行同列数字与上一行前一列数字之和。

实现步骤：

（1）定义数据类型，本实例中 i、j、a[11][11]均为基本整型。

（2）第一个 for 循环中变量 i 的范围为 1～10，循环体中语句 a[i][i]将对角线元素置 1.，语句 a[i][1]=1 将每行中的第一列置 1。

（3）用两个 for 循环实现除对角线和每行第一个元素外其他元素的赋值过程，即 a[i][j]=a[i-1][j-1]+a[i-1][j];

（4）再次 for 循环的嵌套将数组 a 中的所有元素输出。

例 9.10 使用 for 循环嵌套语句，将杨辉三角打印出 10 行，显示在屏幕上。（光盘位置：光盘\MR\Instance\9\10）

```
#include<stdio.h>
main()
{
    int i, j, a[11][11];                                    /*定义 i，j，a[11][11]为基本整型*/
    for (i = 1; i < 11; i++)                                /*for 循环 i 的范围从 1 到 10*/
    {
        a[i][i] = 1;                                        /*对角线元素全为 1*/
        a[i][1] = 1;                                        /*每行第一列元素全为 1*/
}
    /*for 循环范围从第 3 行开始到第 10 行*/
    for (i = 3; i < 11; i++)
    /*for 循环范围从第 2 列开始到该行行数减一列为止*/
        for (j = 2; j <= i - 1; j++)
            /*第 i 行 j 列等于第 i-1 行 j-1 列的值加上第 i-1 行 j 列的值*/
            a[i][j] = a[i - 1][j - 1] + a[i - 1][j];
    for (i = 1; i < 11; i++)                                /*i 的取值范围为 1~10*/
    {
        for (j = 1; j <= i; j++)
            printf("%4d", a[i][j]);                         /*通过上面两次 for 循环将二维数组 a 中元素输出*/
        printf("\n");                                       /*每输出完一行进行一次换行*/
    }
}
```

运行本实例，效果如图 9.10 所示。

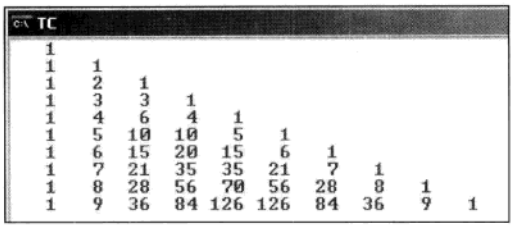


图 9.10　打印出杨辉三角的前 10 行

技巧 11　特殊等式

很多人都说编程是枯燥乏味的，其实不然，只要你真正喜欢这门语言，就会发现很多有趣的东西，例如有这样一个等式，*xyz*+*yzz*=532，编程求 *x*、*y*、*z* 的值（*xyz* 和 *yzz* 分别表示一个三位数）。

实现思路：

本实例的算法思想是对 *x*、*y*、*z* 分别进行穷举，由于 *x* 和 *y* 均可做最高位，所以 *x* 和 *y* 不能为 0，所以穷举范围为 1～9，而 *z* 始终做个位，所以 *z* 的穷举范围为 0～9，对其按照题中要求的等式求和，看和是否等于 532，如果等于，则 *x*、*y*、*z* 就是所求结果，否则继续寻找。

说明：穷举法又称枚举法或列举法，是在研究对象由有限个元素够成的集合时，把所有对象都列举出来，再对其一一进行研究。

例 9.11　利用穷举法，根据给出的特殊等式，求出特殊等式中 *x*、*y*、*z* 的值。（光盘位置：光盘\MR\Instance\9\11）

```
#include<stdio.h>
main()
{
    int x, y, z, i;
    for (x = 1; x < 10; x++)                                    /*对 x 进行穷举*/
        for (y = 1; y < 10; y++)                                /*对 y 进行穷举*/
            for (z = 0; z < 10; z++)                            /*对 z 进行穷举，由于是个位所以可以为 0*/
            {
                i = 100 * x + 10 * y + z + 100 * y + 10 * z + z;  /*求和*/
                if (i == 532)                                   /*判断和是否等于 532*/
                  printf("x=%d,y=%d,z=%d\n", x, y, z);          /*输入 x、y、z 最终的值*/
            }
}
```

运行本实例，效果如图 9.11 所示。

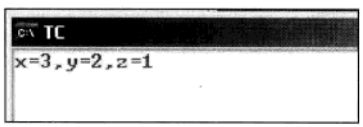


图 9.11　特殊等式

技巧 12 统计各种字符个数

输入一组字符，要求分别统计出其中英文字母、数字、空格以及其他字符的个数。

实现技巧：

（1）实例中定义了一个字符变量 c，以下是关于字符变量的使用要点。

- 字符变量是用来存放字符常量的，一个字符变量只能存放一个字符，不能用来存放一个字符串。
- 字符数据以 ASCII 码存储，实际上是以二进制形式存放的，它的存储形式与整数的存储形式类似，所以一个字符型数据既可以以字符形式输出，也可以以整数形式输出。

（2）使用 getchar()函数给字符变量 c 赋值从而方便对输入字符进行判断。Getchar()函数使用要点如下。

- 一般形式为：getchar()

作用是从终端输入一个字符，函数的值就是从输入设备得到的字符。

- getchar()函数只能接受一个字符。

与其对应的是 putchar()函数，它的作用是向终端输出一个字符。

（3）while((c=getchar())!='\n')，这个 while 的表达式里用到了'\n'，这种字符被称为转义字符，转义字符及其含义如表 9.1 所示。

表 9.1 转义字符

转 义 字 符	转义字符的意义
\n	回车换行
\t	横向跳到下一制表位置
\v	竖向跳格
\b	退格
\r	回车
\f	换页
\\	反斜线符"\"
\'	单引号符
\a	鸣铃
\ddd	1～3 位八进制数所代表的字符
\xhh	1～2 位十六进制数所代表的字符

注意：在使用字符常量的时候应注意使用"将其括起。

例 9.12　执行循环体语句判断输入的字符串中字符的类型，实现统计各种字符个数。（光盘位置：光盘\MR\Instance\9\12）

```
#include<stdio.h>
main()
{
    char c;                                                    /*定义 c 为字符型*/
    /*定义 letters、space、digit、others 4 个变量为基本整型*/
    int letters = 0, space = 0, digit = 0, others = 0;
    printf("please input some characters\n");
    while ((c = getchar()) != '\n')
    /*当输入的不是回车时执行 while 循环体部分*/
    {
        if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
            letters++;                                         /*当输入的是英文字母时变量 letters 加
1*/
        else if (c == ' ')
            space++;                                           /*当输入的是空格时变量 space 加 1*/
        else if (c >= '0' && c <= '9')
            digit++;                                           /*当输入的是数字时变量 digit 加 1*/
        else
            /*当输入的即不是英文字母又不是空格或数字是变量 others 加 1*/
            others++;
    }
    printf("char=%d space=%d digit=%d others=%d\n", letters, space, digit, others);/*将最终统计结果输出
*/
}
```

运行本实例，效果如图 9.12 所示。

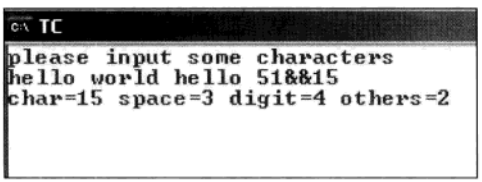


图 9.12　统计各种字符个数

技巧 13　猴子吃桃

猴子吃桃问题：猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个，第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第 10 天早上想再吃时，见只剩下一个桃子了。编写程序求第一天共摘了多少个桃子。

实现思路：

本实例需要先找出变量间的关系，找出了它们之间的关系后程序就基本上没有什么问题了。本题中读者要明确第一天桃子数和第二天桃子数之间的关系即第二天桃子数加 1 的 2 倍等于第一天的

桃子数。

例 9.13 使用 while 语句从后向前推出第一天摘的桃子数。(光盘位置：光盘\MR\Instance\9\13)

```
#include <stdio.h>
main()
{
    int day,x1,x2;                          /*定义 day、x1、x2 3 个变量为基本整型*/
    day=9;                                  /*为变量 day 赋值为 9*/
    x2=1;                                   /*为变量 x2 赋值为 1*/
    while(day>0)
    {
        x1=(x2+1)*2;                                /*第一天的桃子数是第二天桃子数加 1 后的 2 倍*/
        x2=x1;
        day--;                                      /*因为从后向前推天数递减*/
    }
    printf("the total is %d\n",x1);                     /*输出桃子的总数*/
}
```

运行本实例，效果如图 9.13 所示。

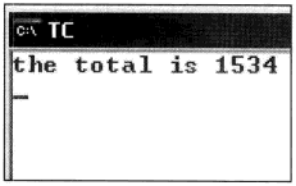


图 9.13 猴子吃桃

技巧 14 删除字符串中连续字符

输入一个字符串，输入要删除的位置及长度，输出完或删除操作后的字符串。

实现思路：

（1）本实例的关键技术是如何实现删除，这里采用的方法是将要删除部分后面字符从要删除部分开始逐个覆盖。

（2）在确定从哪个字符开始删除的时候要尤其注意，例如本实例中字符串 hello world，字母 e 是这个字符串中的第 2 个字符但在数组 s 中它的下标却是 1，也就是说如果用户输入的 position 是 2，那么我们开始进行覆盖的位置在数组 s 中的下标就应是 position-1，同理我们可以确定出将用哪个字符开始覆盖，在本实例中也就是 i 的初值，通过上面的分析我们就很容易确定它在数组中的具体位置，即 postion-1+length。

实现步骤：

（1）自定义删除函数，该函数有 3 个参数。

（2）用 for 语句实现将删除部分后的字符依次从删除部分开始覆盖，最终将新得到的字符串返回。

（3）主函数编写，先定义字符型数组及两个基本整型变量，通过输入函数分别获得字符串、position 及 length 的值，调用 del 函数，最终将新得到的字符串输出。

例 9.14　自定义删除函数，通过输入函数获取字符串的值，实现字符串中连续字符的删除操作。（光盘位置：光盘\MR\Instance\9\14）

```
#include<stdio.h>
char del(char s[],int pos,int len)                    /*自定义删除函数*/
{
    int i;                                            /*定义变量*/
    for(i=pos+len-1;s[i]!='\0';i++,pos++)             /*i 初值为指定删除部分后的第一个字符*/
        s[pos-1]=s[i];                                /*用删除部分后的字符依次从删除部分开始覆盖*/
        s[pos-1]='\0';                                /*在重新得到的字符后加上字符串结束标志*/
    return s;                                         /*返回新得到的字符串*/
}
main()                                                /*main()主函数*/
{
    char str[50];                                     /*定义字符型数组*/
    int position;                                     /*定义整型变量*/
    int length;
    printf("\nPlease input string:");                 /*输出提示信息*/
    gets(str);                                        /*使用 gets 函数获得字符串*/
    printf("\nPlease input delete position:");
    scanf("%d",&position);                            /*输入要删除的位置*/
    printf("\nPlease input delete length:");
    scanf("%d",&length);                              /*输入要删除的长度*/
    del(str,position,length);                         /*调用删除函数*/
    printf("\nThe final string:%s",str);              /*将新得到的字符串输出*/
}
```

运行本实例，效果如图 9.14 所示。

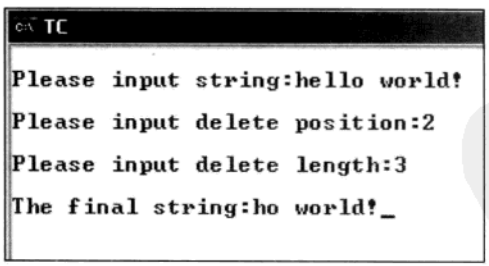


图 9.14　删除字符串中连续字符

技巧 15　求学生的平均身高

输入学生数并逐个输入学生的身高，输出身高的平均值。本技巧可以应用于更多求平均值的地方。

实现思路：

本实例主要采用了数组名作为函数参数，有以下几点需要说明。

（1）用数组名作参数，应该在主调函数和被调用函数中分别定义数组，像本例中主调函数定义的数组为 height，被调用函数定义的数组为 array。

（2）实参数组与形参数组类型应一致，本实例中都为 float。

（3）形参数组也可以不指定大小，在定义数组时在数组名后面跟一个空的方括弧。本实例中没有指定形参数组的大小。

（4）用数组名作函数实参时，不是把数组元素的值传递给形参，而是把实参数组的起始地址传递给形参数组。

实现步骤：

（1）自定义函数 average，用 for 语句实现各个元素值相加，最终求出平均数

（2）主函数编写，输入学生数及每个学生的身高，调用函数 average 求出平均数并将结果输出。

例 9.15 自定义函数 average，用 for 语句实现各个元素值相加，最终求出平均数。（光盘位置：光盘\MR\Instance\9\15）

```
#include<stdio.h>
main()
{
    float average(float array[],int n);                    /*函数声明*/
    float height[100],aver;                                /*定义单精度数组和变量*/
    int i,n;                                               /*定义整型变量*/
    printf("please input the number of students:\n");      /*输出提示信息*/
    scanf("%d",&n);                                        /*输入学生数量*/
    printf("please input student`s height:\n");
    for(i=0;i<n;i++)                                       /*for 循环语句，i 的取值范围小于 n*/
    scanf("%f",&height[i]);                                /*逐个输入学生的身高*/
    printf("\n");
    aver=average(height,n);                                /*调用 average 函数求出平均身高*/
    printf("average height is %6.2f",aver);                /*将平均身高输出*/
}
float average(float array[],int n)                         /*自定义求平均身高函数*/
{
    int i;                                                 /*定义整型变量*/
    float aver,sum=0;
    for(i=0;i<n;i++)
    sum+=array[i];                                         /*用 for 语句实现 sum 累加求和*/
    aver=sum/n;                                            /*总和除以人数求出平均值*/
    return(aver);                                          /*返回平均值*/
}
```

运行本实例，效果如图 9.15 所示。

```
TC
please input the number of students:
5
please input student's height:
166.5
175.3
169.2
174.6
166.8

average height is 170.48
```

图 9.15　求出学生的平均身高

技巧 16　灯塔数量

有一个 8 层灯塔，每层的灯数都是上一层的一倍，共有 765 盏灯，编程求最上层与最下层的灯数。

实现思想：

本实例没有太多难点，通过对 n 的穷举，探测满足条件的 n 值。在计算灯的总数时先计算 2 楼到 8 楼灯的总数，再将计算出的和加上 1 楼灯的数量，这样就求出了总数，当然也可以将一楼灯的数量赋给 sum 之后再加上 2 楼到 8 楼灯的数量。

例 9.16　利用 while 循环对 n 的值从 1 开始进行穷举，求出最上层和最下层的灯塔数量。（光盘位置：光盘\MR\Instance\9\16）

```
#include<stdio.h>
main()
{
    int n = 1, m, sum, i;                          /*定义变量为基本整型*/
    while (1)
    {
        m = n;                                     /*m 存储一楼灯的数量*/
        sum = 0;
        for (i = 1; i < 8; i++)
        {
            m = m * 2;                             /*每层楼灯的数量是上一层的 2 倍*/
            sum += m;                              /*计算出除一楼外灯的总数*/
        }
        sum += n;                                  /*加上一楼灯的数量*/
        if (sum == 765)                            /*判断灯的总数量是否达到 765*/
        {
            printf("the first floor has %d\n", n);  /*输出一楼灯的数量*/
            printf("the eight floor has %d", m);    /*输出八楼灯的数量*/
            break;                                 /*跳出循环*/
        }
        n++;                                       /*灯的数量加 1，继续下次循环*/
    }
}
```

运行本实例，效果如图 9.16 所示。

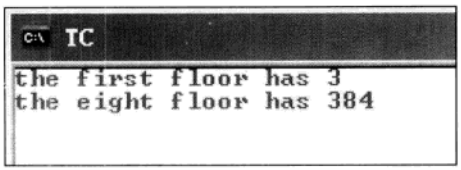


图 9.16　最上层和最下层灯塔数量

技巧 17　老师分糖果

幼儿园老师将糖果分成了若干等份，让学生按任意次序上来领，第 1 个来领的，得到 1 份加上剩余糖果的十分之一；第 2 个来领的，得到 2 份加上剩余糖果的十分之一；第 3 个来领的，得到 3 份加上剩余糖果的十分之一，……问共有多少个学生，老师共将糖果分成了多少等份？

实现思想：

读者在看到题目时也许感觉无从下手，这里就给大家介绍一个方法，即采用穷举法进行探测，由部分推出整体。设老师共将糖果分成 n 等份，第 1 个学生得到的份数为 sum1=（n+9）/10，第 2 个学生得到的份数为 sum2=（9*n+171）/100，为 n 赋初值，本实例中 n 初值赋 11（糖果份数至少为 11 份时，第一个来领的同学领到的才是完整的份数），穷举法直到 sum1=sum2，这样就可以计算出老师将糖果分成了多少份和学生的数量。

实现步骤：

（1）定义 n 为基本整型，sum1 和 sum2 为单精度型。

（2）使用穷举法，这里用 for 语句对 n 逐个判断，直到满足条件 sum1=sum2，结束 for 循环。这里有一个问题值得大家注意，因为将糖果分成了 n 等份，所以最终求出的结果必须是整数。

（3）将最终求出的结果输出，这里注意一下学生数量是用总份数除以每个人得到的份数，程序里是除以第一个人得到的份数。

例 9.17　使用穷举法，用 for 语句进行逐个判断，求出老师共将糖果分成了多少等份。(光盘位置：光盘\MR\Instance\9\17)

```
#include <stdio.h>
main()
{
    int n;                                          /*定义整型变量*/
    float sum1,sum2;                                /*sum1 和 sum2 应为单精度型，否则结果将不准确*/
    for(n=11;;n++)                                  /*for 循环语句中没有第二个条件*/
    {
        sum1=(n+9)/10.0;                            /*将公式求出的结果赋值给 sum1*/
        sum2=(9*n+171)/100.0;                       /*将公式求出的结果赋值给 sum2*/
        /*sum1 和 sum2 应为整数，否则结束本次循环继续下次判断*/
        if(sum1!=(int)sum1)
            continue;
        if(sum2!=(int)sum2)
```

```
            continue;
        if(sum1==sum2) break;                          /*当 sum1 等于 sum2 时，跳出循环*/
    }
    /*输出学生数及分成的份数*/
    printf("\n the number of students is %d\n gong fen le %d fen",(int)(n/sum1),n);
}
```

运行本实例，效果如图 9.17 所示。

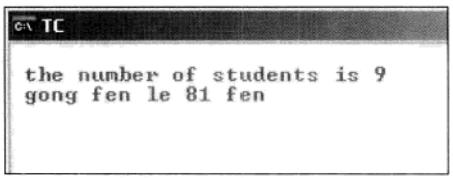


图 9.17　老师分糖果

技巧 18　小球下落问题

一球从 100 米高度自由落下，每次落地后反跳回原高度的一半；再落下，求它在第 10 次落地时，共经过多少米？第 10 次反弹多高？

实现思路：

想解决本题最主要是要分析小球每次弹起的高度与落地次数之间的关系。这里给大家分析下：小球从高处自由下落当第一次落地时经过了 100 米，这个可单独考虑，从第一次弹起到第二次落地前经过的路程为前一次弹起最高高度的一半乘以 2 加上前面经过的路程，因为每次都有弹起和下落两个过程其经过的路程相等故乘以 2。

以后的几次依此类推，那么到第十次落地前共经过了九次这样的过程，所以程序中 for 循环执行循环体的次数是九次。题目中还提到了第十次反弹的高度，这个我们只需在输出时用第九次弹起的高度除以 2 即可。

以后遇到此类问题时不要急着先写程序，应先理清题目思路找出其中规律，这时再写程序会更加得心应手。

实现步骤：

（1）定义 i，h，s 为单精度型并为 h、s 赋初值 100。

（2）使用 for 语句计算每一次弹起到落地所经过的路程与前面经过路程和的累加。

（3）将经过的总路程及第十次弹起的高度输出。

例 9.18　分析题目中潜藏的规律，使用 for 循环语句，求出小球下落问题。（光盘位置：光盘\MR\Instance\9\18）

```
#include <stdio.h>
main()
{
    /*定义变量 i，h，s 分别为单精度型并为 h 和 s 赋初值 100*/
    float i,h=100,s=100;
    /*for 语句，i 的范围从 1 到 9 表示小球从第二次落地到第十次落地*/
    for(i=1;i<=9;i++)
    {
        h=h/2;                                          /*每落地一次弹起高度变为原来一半*/
        /*累积的高度和加上下一次落地后弹起与下落的高度*/
        s+=h*2;
    }
    printf("the total length is %f\n",s);               /*将高度和输出*/
    printf("the height of tenth time is %f",h/2);       /*输出第十次落地后弹起的高度*/
}
```

运行本实例，效果如图 9.18 所示。

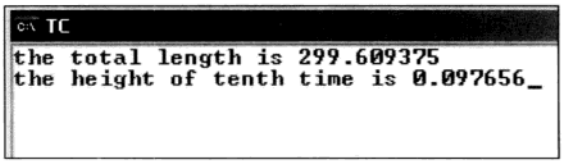


图 9.18　小球下落问题

技巧 19　彩球问题

在一个袋子里装有三色彩球，其中红色球有 3 个，白色球也有 3 个，黑色球有 6 个，问当从袋子中取出 8 个球时，共有多少种可能的方案。编程实现将所有可能的方案编号输出在屏幕上。

实现思路：

本实例和百钱百鸡问题在解题思路上基本相同，都是要先确定范围，本实例要确定各种颜色球的范围，红球和白球的范围根据题意可知均是大于等于 0 小于等于 3，不同的是本实例将黑球的范围作为 if 语句中的判断条件，即用要取出的球的总数目 8 减去红球及白球的数目所得的差应小于黑球的总数目 6。

实现步骤：

（1）定义 i，j，count 分别为基本整型，count 赋初值为 1，这里起到计数的功能。

（2）使用 for 语句进行穷举，对满足 if 语句中条件的可能方案按指定格式进行输出，否则进行下次循环。

例 9.19　使用 for 循环语句和 if 选择判断语句解决彩球问题。(光盘位置：光盘\MR\Instance\9\19)

```
#include <stdio.h>                                      /*引用头文件*/
main()
{
    int i, j, count;                                    /*定义整型变量*/
```

```
    puts("the result is:\n");                                    /*输出提示信息*/
    printf("time  red ball  white ball  black ball\n");
    count = 1;
    for (i = 0; i <= 3; i++)                                     /*红球数量范围 0～3 之间*/
        for (j = 0; j <= 3; j++)                                 /*白球的数量范围 0～3 之间*/
            /*判断要取黑色球的数量是否在 6 个以内*/
            if ((8-i - j) <= 6)
                printf("%3d%8d%9d%10d\n", count++, i, j, 8-i - j);   /*输出各种颜色球的数量*/
    return 0;
}
```

运行本实例，效果如图 9.19 所示。

```
c:\ TC
the result is:

time  red ball  white ball  black ball
   1        0          2          6
   2        0          3          5
   3        1          1          6
   4        1          2          5
   5        1          3          4
   6        2          0          6
   7        2          1          5
   8        2          2          4
   9        2          3          3
  10        3          0          5
  11        3          1          4
  12        3          2          3
  13        3          3          2
```

图 9.19　取出彩球的可能方案



第 10 章　指针开发技巧

通过学习本章，用户可以掌握常用的指针开发技巧，包括实现数据交换，实现整数排序，指向结构体变量的指针，输出数组元素，实现逆序存放数组元素值，查找数列中最大值最小值，比较大小，实现字符串复制、连接、插入、匹配，实现输入月份号输出该月份英文名等。

技巧 01　使用指针实现整数排序

用指针实现输入 3 个整数，将这 3 个整数按照由大到小的顺序输出，显示在屏幕上。

实现思路：

本技巧用到了函数的嵌套调用，自定义函数 swap，用来实现两个数的互换；自定义函数 exchange 用来完成 3 个数的位置交换，它的内部嵌套使用了 swap 自定义函数。这两个函数的参数都是指针变量，实现了传址的功能，即改变形参的同时，实参也被改变。

实现步骤：

（1）创建自定义函数 swap()，用来实现数据的交换。

（2）创建 exchange()自定义函数，用于比较数值大小，并调用 swap()自定义函数，交换数据的位置。

（3）创建 main()函数作为程序的入口程序，并在该函数中调用 exchange()函数，实现对输入的 3 个数据比较大小并交换位置。

例 10.1　创建自定义函数，使用指针实现对输入的整数按照由大到小的顺序排序。（光盘位置：光盘\MR\Instance\10\01）

```
#include<stdio.h>
swap(int *p1, int *p2)                          /*交换两个数据*/
{
    int temp;                                   /*声明整型变量，用于存储数据*/
    temp=*p1;                                   /*实现数据交换*/
    *p1=*p2;
    *p2=temp;
}
exchange(int *pt1, int *pt2, int *pt3)          /*比较数值大小并进行交换*/
{
    /*从大到小顺序排列*/
    if (*pt1 <*pt2)                             /*如果 pt1 小于 pt2*/
```

```
        swap(pt1, pt2);                                    /*调用 swap()函数，交换 pt1 与 pt2 的值*/
    if (*pt1 <*pt3)                                        /*如果 pt1 小于 pt3*/
        swap(pt1, pt3);                                    /*调用 swap()函数，交换 pt1 与 pt3 的值*/
    if (*pt2 <*pt3)                                        /*如果 pt2 小于 pt3*/
        swap(pt2, pt3);                                    /*调用 swap()函数，交换 pt2 与 pt3 的值*/
}
main()                                                     /*主函数*/
{
    int a,b,c,*q1,*q2,*q3;                                 /*声明变量*/
    puts("Please input three key numbers you want to rank:");
    scanf("%d,%d,%d", &a, &b, &c);                         /*输入 3 个数*/
    q1 = &a;
    q2 = &b;
    q3 = &c;
    exchange(q1,q2,q3);                                    /*比较大小交换位置*/
    printf("\n%d,%d,%d\n",a,b,c);                          /*输出交换后的值*/
    puts("\n Press any key to quit...");
    getch();
}
```

运行本实例，效果如图 10.1 所示。

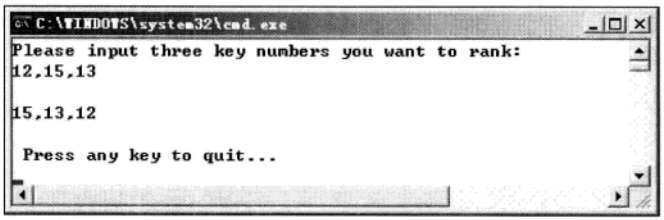


图 10.1　使用指针实现整数排序

技巧 02　使用指针实现数据交换

在实际应用中常常需要进行数据交换，在某些特定的时候还不能直接应用数值交换的方法，本技巧实现使用指针变量实现交换两个变量（a 和 b）的值。运行后，输入两个整型数值，将变量 a，b 中的值交换，然后输出到窗体上。

实现思路：

本程序创建了一个自定义函数 swap，用于实现交换两个变量的值。swap 函数包括两个指针型的形参 p1、p2。在主函数中定义了两个指针型的实参 pointer1 和 pointer2。在函数调用时，将实参变量的值传递给形参变量。交换完成后，P1 和 pointer1 都指向变量 a，p2 和 pointer2 都指向 b。在主函数中输出的变量 a 和变量 b 的值是已经交换过的。

例 10.2　创建自定义函数 swap()，用来实现数据的交换。（光盘位置：光盘\MR\Instance\10\02）

```
#include<stdio.h>
swap(int *p1, int *p2)
{
    int temp;                                              /*声明整型变量*/
    temp =*p1;                                             /*交换两个指针指向的值*/
```

```
    *p1 =*p2;
    *p2 = temp;
}
main()                                          /*main()主函数*/
{
    int a, b;                                   /*定义整型变量*/
    int *pointer1,*pointer2;                    /*声明两个指针变量*/
    scanf("%d,%d", &a, &b);                     /*输入两个数*/
    pointer1 = &a;
    pointer2 = &b;
    swap(pointer1, pointer2);                   /*调用交换函数*/
    printf("\nThe result is :%d,%d\n", a, b);   /*输出交换后的结果*/
    getch();
}
```

运行本实例，效果如图 10.2 所示。

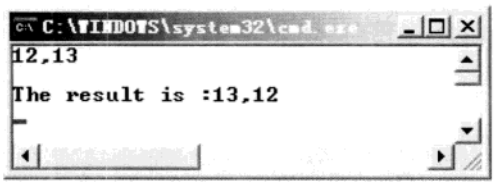


图 10.2　使用指针实现数据交换

技巧解析：

本实例利用指针变量实现数据的交换。变量的指针就是变量的地址，存放地址的变量就是指针变量，用来指向另一个变量。在程序中使用一个“*”表示“指向”，定义指针变量的一般形式为：

```
基类型  *指针变量名
```

例如下面定义指针变量的语句都是正确的。

```
int *p;
char *s;
float *lp;
```

因为指针变量是指向一个变量的地址，所以将一个变量的地址值赋给这个指针变量后，这个指针变量就“指向”了该变量。例如，将变量 i 的地址存放到指针变量 p 中，p 就指向 i。其关系如图 10.3 所示。

指针变量前面的“*”必不可少，表示该变量的类型为指针型变量。指针变量的名称为 p。

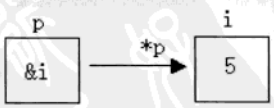


图 10.3　p、*p 和 i 的关系

本程序创建了一个自定义函数 swap，用于交换两个变量的值。swap 函数包括两个指针型的形参 p1、p2。在主函数中定义了两个指针型的实参 pointer1 和 pointer2。在函数调用时，将实参变量的值传递给形参变量。交换完成后，P1 和 pointer1 都指向变量 a，p2 和 pointer2 都指向 b。在主函数中输出的变量 a 和变量 b 的值是已经交换过的。

注意：定义指针变量时必须指定基类型。因为要根据指定的类型决定分配的空间。例如，定义指针类型为整型，当指针移动一个位置时，其地址值加 2；如果指针指向一个实型变量，则增加值为 4。

技巧 03　指向结构体变量的指针

一个结构体变量的指针就是该变量所占据的内存段的起始地址。用一个指针变量指向一个结构体变量，此时该指针变量的值是结构体变量的起始地址。本技巧通过结构体指针变量实现在窗体上显示学生信息。

实现步骤：

（1）声明 struct student 类型。

（2）创建 main()函数作为程序的入口程序，在 main()函数中定义 struct student 类型的变量。定义一个指针变量指向 struct student 类型的数据。

例 10.3　用指针变量指向结构体变量，实现在窗体上显示学生信息。（光盘位置：光盘\MR\Instance\10\03）

```
#include<stdio.h>
struct student
{
    int num;                                    /*学生学号*/
    char name[20];                              /*学生姓名*/
    char sex;                                   /*学生性别*/
    int age;                                    /*学生年龄*/
    float score;                                /*学生成绩*/
};                                              自定义结构体 struct student
void main()
{
   struct student student1={1001,"liming",'M',20,92.5};   /*定义结构体变量*/
   struct student *p;                           /*定义指针变量指向结构体类型*/
   p=&student1;                                 /*使指针指向结构体变量*/
   printf("Number:%d\n",p->num);                /*输出学生学号*/
   printf("Name:%s\n",p->name);                 /*输出学生姓名*/
   printf("Sex:%c\n",p->sex);                   /*输出学生性别*/
   printf("Age:%d\n",p->age);                   /*输出学生年龄*/
   printf("Score:%f\n",p->score);               /*输出学生成绩*/
   getch();
}
```

运行本实例，效果如图 10.4 所示。

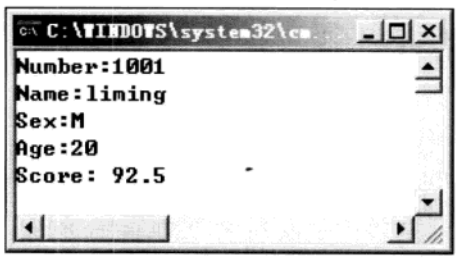


图 10.4　显示学生信息

技巧 04 使用指针输出数组元素

本实例通过指针变量输出数组的各元素值，运行程序后，输入 10 个数值，可以看到输出的数组元素值。

例 10.4 创建 main()函数，在 main()函数中将用户输入的 10 个数值存入到数组中，使用指针变量指向这个数组，并实现数组元素的输出。（光盘位置：光盘\MR\Instance\10\04）

```
#include<stdio.h>
main()
{
    int a[10];                                  /*定义整型数组*/
    int *p,i;                                   /*定义指针和变量*/
    puts("\nPlease input ten integer:\n");      /*输出提示信息*/
    for (i = 0;i<10;i++)                        /*i 的范围为 0~9*/
        scanf("%d",&a[i]);                      /*输入数组元素*/
    printf("\n");
    for (p=&a;p<(a + 10);p++)                   /*使用指针指向数组，并输出数组元素*/
        printf("%d",*p);
    puts("\n\n Press any key to quit...");
    getch();
    return 0;
}
```

运行本实例，效果如图 10.5 所示。

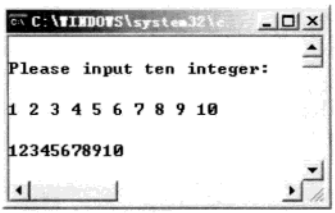


图 10.5 使用指针输出数组元素

技巧解析：

本实例应用指向数组的指针实现数组元素的输出。定义一个指向数组元素的指针变量的方法，与定义指向变量的指针变量相同。例如：

```
int a[10];    /*定义一个包含 10 个整型数据的数组*/
int *p;       /*定义整型指针变量 p*/
p=&a[0];      /*对指针变量赋值*/
```

上面第 3 句代码把数组 a 的 a[0]元素的地址赋给指针变量 p。也就是说，p 指向 a 数组的首元素，如图 10.6 所示。指针变量 p 中存放 a[0]的地址，因此 p 指向 a[0]。C 语言中数组名代表数组的首地址，也就是数组第一个元素的地址。因此，下面两个语句等价：

```
p=&a[0];
p=a;
```

使用指针指向数组之后，就要通过指针引用数组元素。在 C 语言中，如果指针变量 P 指向数组中的一个元素，则 P+1 指向同一数组中的下一个元素，如图 10.7 所示。如果 p 的初值为&a[0]，则 p+1 的值就是&a[1]，也可表示为 a+1。同理，p+i 和 a+i 就是 a[i]的地址，或者说，它们都指向 a 数组的第 i 个元素。则*（p+i）就是 p+i 所指向的数组元素，即 a[i]。

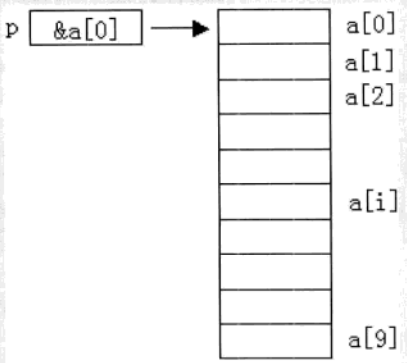


图 10.6　指向数组的指针

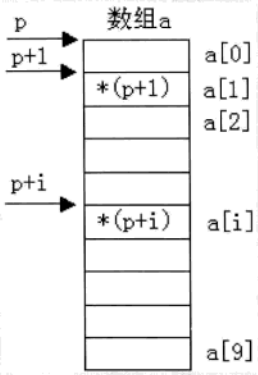


图 10.7　通过指针引用数组元素

本程序使用指针变量指向一个数组，使 P 的初始值为数值元素首地址。使用 P+1 移动指针，使指针指向数组的每个元素值。

技巧 05　用指针实现逆序存放数组元素值

在数组元素中存放一般为 a[0]，a[1]，a[2]，……，a[n]的顺序，本技巧是使用指针实现逆序存放数组元素，即存放顺序为 a[n]，a[n－1]，……，a[1]，a[0]。

实现思想：

本程序可以创建一个函数 inverte()用来实现对数组元素的逆序存放。自定义函数的形参为一个指向数组的指针变量 x，x 初始值指向数组 a 的首元素的地址。x+n 是 a[n]元素的地址。声明指针变量 i、j 和 p，i 初值为 x，即指向数组首元素地址，j 的初值为 x+n-1，即指向数组最后一个元素地址，使 p 指向数组中间元素地址。交换*i 与*j 的值，即交换 a[i]与 a[j]的值。移动 i 和 j，使 i 指向数组第二个元素，j 指向倒数第二个元素，继续交换，直到中间值。这样就实现了数组元素的逆序存放。

例 10.5　创建自定义函数 inverte()用来实现对数组元素的逆序存放。（光盘位置：光盘\MR\Instance\10\05）

引用头文件。

```
#include<stdio.h>
```

创建自定义函数 inverte()用来实现对数组元素的逆序存放。

```
inverte(int *x, int n)
{
    int *p, temp,  *i,  *j, m = (n - 1) / 2;      /*声明变量*/
    i = x;                                        /*变量 i 存放数组首地址*/
    j = x + n - 1;                                /*变量 j 存放数组末尾元素地址*/
    p = x + m;                                    /*变量 p 存放数组中间元素地址*/
    for (; i <= p; i++, j--)                      /*交换数组前半部分和后半部分元素*/
    {
        temp =*i;
         *i =*j;
         *j =temp;
    }
    return 0;
}
```

创建 main()函数，在函数中调用 inverte()函数，并将逆序后的数组输出显示在窗体上。实现代码如下：

```
void main()                                                    /*main()主函数*/
{
    /*void inverte(int *x,int n);*/
    int i, a[10] =
    {
        1, 2, 3, 4, 5, 6, 7, 8, 9, 0
    };                                                         /*定义数组*/
    printf("The elements of original array:\n");
    for (i = 0; i < 10; i++)                                       /*输出数组*/
        printf("%d,", a[i]);
    printf("\n");                                                /*换行*/
    inverte(a, 10);                                                /*使数组元素逆序*/
    printf("The elements has been inverted:\n");                 /*输出提示信息*/
    for (i = 0; i < 10; i++)                                       /*输出逆序后数组*/
        printf("%d,", a[i]);
    printf("\n");                                                /*换行*/
    getch();
}
```

运行本实例，效果如图 10.8 所示。

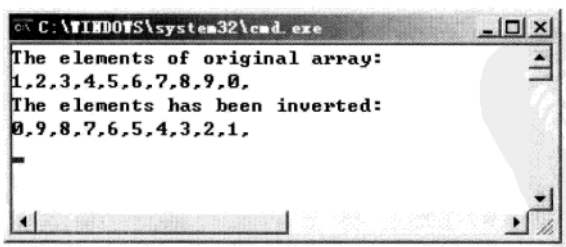


图 10.8　数组的逆序存放

技巧 06　使用指针查找数列中最大值/最小值

输入 10 个整数，找出最大值和最小值，相信通过前面的学习已经可以轻松实现。但是有些特

殊的情况或原因，要求必须使用指针查找出数列中最大值和最小值，本技巧正是教你如何使用指针实现查找数列中的最大值和最小值。

实现思路：

本技巧使用指向一维数组的指针遍历一维数组中数据，从而实现查找数组中的最大值/最小值。

在本技巧中，自定义函数 max_min()用于将求得的最大值和最小值分别存放在这两个变量 max 和 min 中。变量 max 和 min 是在 main()函数中定义的局部变量，将这两个变量的地址作为函数参数传递给被调用函数 max_min()，函数执行后将数组中最大值和最小值存储在 max 和 min 中并返回，这就是数值的传递过程。

下面介绍如何查找数组中最大值和最小值。在自定义函数 max_min()中，定义了指针变量 p 指向数组，其初值为 a+1，也就是使 p 指向 a[1]。循环执行 p++，使 p 指向下一个元素。每次循环都将*p 和*max 与*min 比较，将大值存放在 max 所指地址中，将小值存放在 min 所指地址中。

例 10.6　创建 max_min()自定义函数实现查找数组中最大值和最小值。创建 main()函数，在此函数中调用 max_min()函数，并将所得结果输出在窗体上。（光盘位置：光盘\MR\Instance\10\06）

引用头文件。

```
#include<stdio.h>
```

创建 max_min()自定义函数实现查找数组中最大值和最小值的关键代码。

```
void max_min(int a[], int n, int *max, int *min)          /*自定义 max_min()函数*/
{
    int *p;                                               /*定义指针*/
    *max = *min = *a;                                     /*初始化最大值最小值指针变量*/
    for (p = a + 1; p < a + n; p++)                       /*for 循环语句*/
        if (*p >*max)
            *max = *p;                                    /*最大值*/
        else if (*p < *min)
            *min = *p;                                    /*最小值*/
    return 0;
}
```

创建 main()函数，在此函数中调用 max_min()函数，并将所得结果输出在窗体上。实现代码如下：

```
main()                                                    /*创建 main()主函数*/
{
    int i, a[10];                                         /*定义整型变量和数组*/
    int max, min;                                         /*定义整型变量*/
    printf("Input 10 integer numbers you want to operate:\n ");
    for (i = 0; i < 10; i++)
        scanf("%d", &a[i]);                               /*输入数组元素*/
    max_min(a, 10, &max, &min);                           /*返回最大值和最小值*/
    printf("\nThe maximum number is: %d\n", max);         /*输出最大值*/
    printf("The minimum number is: %d\n", min);           /*输出最小值*/
    getch();
}
```

运行本实例，效果如图 10.9 所示。

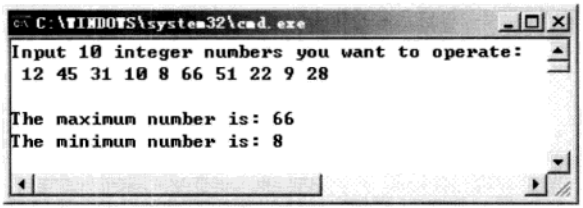


图 10.9　使用指针查找数列中最大值/最小值

技巧 07　使用指针实现字符串复制

本技巧实现使用指针实现字符串的复制，并将字符串输出到窗体上。

实现思想：

本实例用字符型指针变量和指向字符串的指针做函数的参数实现字符串的复制。在主函数中定义了两个字符型指针变量，指针变量 str 用于指向要被复制的字符串，指针变量 p 用于指向复制后的字符串。

例 10.7　创建一个自定义函数，用于实现字符串复制，在主函数中调用自定义函数，将结果显示窗体上，关键代码如下。(光盘位置：光盘\MR\Instance\10\07)

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，声明自定义函数。

```
#include <stdio.h>
copy(char *s,char *q);
```

（3）创建自定义函数 copy()，实现字符串的复制。

```
copy(char *s, char *q)                          /*实现字符串复制*/
{
    int i = 0;                                  /*定义整型变量并进行初始化*/
    for (;*s!= '\0';)
    {
        *q =*s;
        s++;
        q++;
    }
    *q = '\0';                                  /*添加字符串结束符*/
}
```

（4）创建 main()函数，在此函数中调用 copy()函数，将得到的结果输出在窗体上。实现代码如下：

```
int main(void)                                  /*main()主函数*/
{
```

```
    char *str,  *p;                          /*声明字符型指针变量*/
    str = "Hello world!";
    copy(str, p);                          /*实现复制*/
    printf("%s", p);                         /*输出复制后字符串*/
    getch();
    return 1;
}
```

运行本实例，效果如图 10.10 所示。

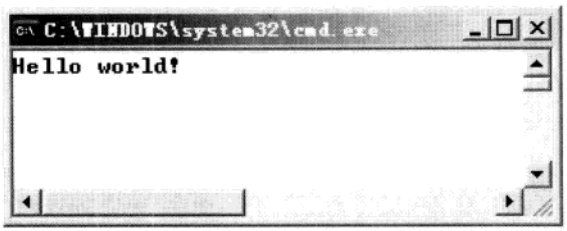


图 10.10　使用指针实现字符串复制

技巧解析：

在 C 语言中对字符串是按字符数组处理的，在内存开辟一个字符数组用来存放该字符串。使用字符串对字符指针变量 str 初始化，实际上是把字符串的第一个元素的地址赋给 str，而不是把整个字符串里的几个字符赋给该字符串变量。图 10.11（a）中所示 string 为一个字符串数组，10.11（b）中所示为一个指针指向的字符串。定义字符串指针变量可写作如下形式：

```
char *str="Hello world!"
```

也可以写做下面两行的格式：

```
char *str;
str="Hello world!";
```

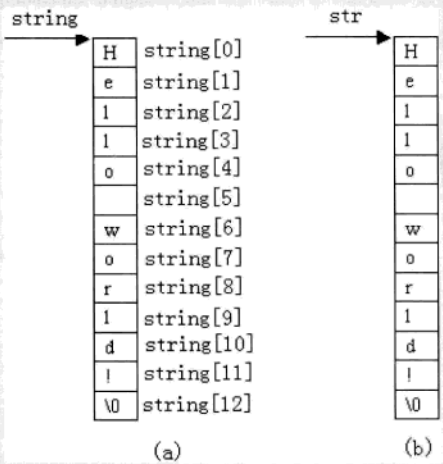


图 10.11　字符串数组和指针指向的字符串

本实例中自定义函数 copy()用于实现字符串的复制。它的两个形参 s 和 q 是字符型指针变量。在 for 循环中，当*s 不为 '\0' 时，每次将*s 赋给*q，然后使 s 和 q 增值，从而实现字符串的复制。

技巧 08 使用指针实现字符串的连接

本实例使用指针实现字符串的连接，并将字符串输出到窗体上。

实现思想：

本技巧用字符型指针变量和指向字符串的指针做函数的参数实现字符串的连接。

本技巧创建了自定义函数 connect()，用于实现字符串的连接。Connect()函数带有 3 个参数，两个用于连接的字符串，一个作为连接后的字符串。

例 10.8 创建自定函数 connect()，用于实现字符串的连接，在 main()主函数中调用自定义函数 connect()，并将连接后的字符串输出在窗体上。（光盘位置：光盘\MR\Instance\10\08）

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，声明自定义函数。

```
#include <stdio.h>
connect(char *s,char *t,char *q);
```

（3）创建自定义函数 connect()，实现字符串的连接。

```
connect(char *s, char *t, char *q)
{
    int i = 0;
    for (;  *s != '\0';)                          /*放入第一个字符串*/
    {
        *q =*s;
        s++;
        q++;
    }
    for (;*t != '\0';)                            /*连接第二个字符串*/
    {
        *q =*t;
        t++;
        q++;
    }
    *q = '\0';
}
```

（4）创建 main()函数，在此函数中调用 connect()函数，将得到的结果输出在窗体上。实现代码如下：

```
int main(void)
{
    char *str,*t,*p;                              /*定义指针*/
    str ="One world";                             /*用于连接的字符串*/
```

```
    t ="one dream";                          /*用于连接的字符串*/
    printf("The first string is:%s\n",str);  /*输出提示信息*/
    printf("The second string is:%s\n",t);   /*输出提示信息*/
    connect(str,t,p);                        /*将两个字符串连接*/
    printf("The connected string is:\n");    /输出提示信息*/
    printf("%s",p);                          /*输出连接后的字符串*/
    getch();
}
```

运行本实例，效果如图 10.12 所示。

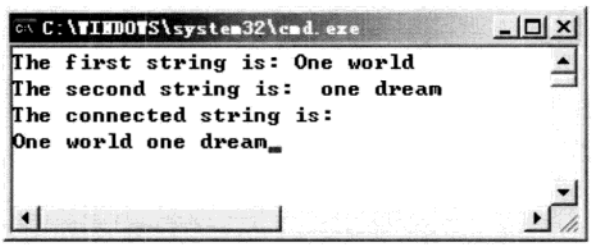


图 10.12　使用指针实现字符串的连接

技巧 09　使用指针实现字符串插入

本实例将一个字符串插入到另一个字符串的指定位置，构成一个新的字符串，并输出到窗体上。

实现思路：

本技巧使用指向字符串的指针实现将一个字符串插入到另一个字符串的指定位置。

本技巧中的字符串插入与字符串的复制实例设计思想基本相同，就是使用指向字符串的指针实现字符串复制的过程，不同的是，字符串的复制是将要复制的字符串放在原有字符串的后面，而字符串的插入是将要插入的字符串插入到指定的位置。

例 10.9　创建自定义函数 insert()，用于实现字符串的插入操作，在 main()主函数中调用自定函数 insert()，将字符串插入的结果输出在窗体上。(光盘位置：光盘\MR\Instance\10\09)

（1）在 TC 下创建一个 C 文件。

（2）引用头文件，声明自定义函数。

```
#include <stdio.h>
#include<string.h>
insert(char *s,char *q,int i);
```

（3）创建自定义函数 insert()，实现字符串插入操作。

```
insert(char s, char q, int n)
{
    int i = 0;                               /*声明整型变量 i*/
    char *str, strcp[50];
    str = strcp;                             /*把数组首地址赋给指针变量*/
```

```
    for (i = 0;  *s != '\0'; i++)                    /*for 循环语句*/
    {
        if (i == n - 1)                              /*对 i 的值进行判断*/
        {
            for (;  q != '\0';)                      /*循环插入字符串*/
            {
                str[i] =  *q;                        /*插入字符*/
                q++;                                 /*变量 q 的值加 1*/
                i++;                                 /*变量 i 的值加 1*/
            }
        }
        str[i] =  *s;
        s++;                                         /*变量 s 加 1*/
    }
    str[i + 1] = '\0';
    return str;                                      /*返回数组首地址*/
}
```

（4）创建 main()函数，在此函数中调用 insert()函数，将得到的结果输出在窗体上。实现代码如下：

```
int main(void)
{
    char *strin,*str;                                /*定义字符串指针*/
    int i;                                           /*定义整型变量*/
    str = "Hello world!";
    strin = "big ";
    printf("Please input the positon you want to insert:");   /*输出提示信息*/
    scanf("%d", &i);                                 /*输入插入位置*/
    str = insert(str, strin, i);                     /*在指定位置插入*/
    printf("\n%s", str);                             /*输出插入后的字符串*/
    getch();
}
```

运行本实例，效果如图 10.13 所示。

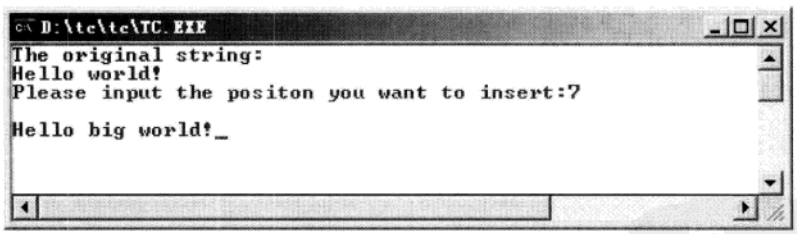


图 10.13　使用指针实现字符串插入

技巧 10　使用指针实现字符串的匹配

本技巧实现对两个字符串进行匹配操作，即在第一个字符串中查找是否存在第二个字符串。如果字符串完全匹配，则提示匹配的信息，并显示第二个字符串在第一个字符串中的开始位置，否则提示不匹配。

实现思想：

本实例定义了自定义函数 match()进行字符串的匹配操作。match()函数包含两个参数，参数

B 为要进行匹配操作的字符串，其类型为字符型指针；参数 A 为用来匹配的字符串。使用循环语句比较 A 字符串最后一个字符是否与 B 中字符相同，如果相同，使用循环语句比较 A 与 B 是否匹配。

例 10.10　自定义函数 match()，进行字符串的匹配操作，在 main()主函数中调用自定义函数 match()，并将字符串匹配的结果输出显示在窗体中。(光盘位置：光盘\MR\Instance\10\10)

（1）在 TC 下创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>
#include <string.h>
```

（3）创建自定义函数 match()，实现字符串匹配操作。

```
int  match( char  B,  char  A )                          /*此函数实现字符串的匹配操作*/
{
    int i, j, start= 0;                                  /*声明变量*/
    int lastB = strlen (B)-1;
    int lastA = strlen (A)-1 ;
    int endmatch =lastA;
    for(j=0;endmatch<=lastB;endmatch++,start++)          /*for 循环语句对字符串进行匹配*/
    {
        if ( B[endmatch] == A[lastA])
            for (j=0,i=start;j<lastA&&B[i]==A[j];)
            i++,j++;
        if ( j == lastA)                                 /*判断 j==last A*/
        {
            return (start+1);                            /*成功  */
         }
    }
    if(endmatch>lastB)                                   /*循环输出*/
    {
        printf("The string is not matchable!");
        return -1;
    }
}
```

（4）创建 main()函数，在此函数中调用 match()函数，将得到的结果输出在窗体上。实现代码如下：

```
void main()                                              /*main()主函数*/
{
    char s[] = "One world,one dream";                    /*原字符串*/
    char t[] = "world";                                  /*要测试匹配的字符串*/
    int p = match(s,t);
    if (p !=-1)                                          /*如果匹配成功输出位置*/
    {
        printf("Matchable!\n");
        printf("The start position is %d", p);
    }
    printf("\n");                                        /*换行*/
```

```
    getch();
}
```

运行本实例，效果如图 10.14 所示。

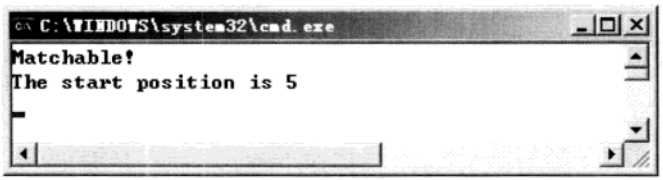


图 10.14　使用指针实现字符串的匹配

技巧 11　用指向函数的指针比较大小

本技巧实现输入两个整数后，将输入的较小值输出显示在窗体上。本技巧使用指向函数的指针实现调用比较数值大小的函数。一个函数在编译时被分配一个入口地址，这个地址就称为函数的指针。所以也可以使用指针变量指向一个函数，然后通过该指针变量调用这个函数。

例 10.11　使用指向函数的指针实现调用比较数值大小的函数，实现用指向函数的指针比较大小。（光盘位置：光盘\MR\Instance\10\11）

引用头文件：

```
#include<stdio.h>
```

创建自定义函数 min()实现返回输入的两个整数中的较小值。

```
min(int a ,int b)                                    /*自定义函数*/
{
    if(a<b) return a;                                /*如果 a 小于 b 则返回 a*/
    else return b;                                   /*否则返回 b*/
}
```

创建 main()函数，在此函数中实现定义指向函数的指针变量，并使此函数指针变量指向 min()函数，将得到的结果输出在窗体上。实现代码如下：

```
void main()                                          /*main()主函数*/
{
    int(*pmin)();                                    /*定义指针*/
    int a, b, m;                                     /*定义 3 个整型变量*/
    pmin = min;
    printf("Please input two integer numbers: \n");  /*输出提示信息*/
    scanf("%d%d", &a, &b);                           /*输入两个值*/
    m = (*pmin)(a, b);                               /*返回最小值*/
    printf("min=%d", m);
    getch();
}
```

运行本实例，效果如图 10.15 所示。

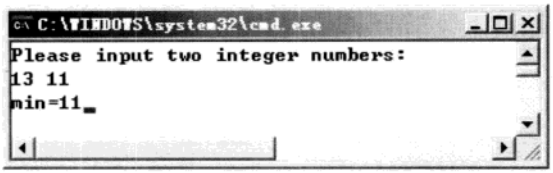


图 10.15　用指向函数的指针比较大小

技巧解析：

指向函数的指针变量的一般形式为：

```
数据类型 (*指针变量名)();
```

这里的数据类型是指函数返回值的类型。

例如：

```
int (*pmin)();
```

(*p)()表示定义一个指向函数的指针变量，它用来存放函数的入口地址，在程序设计过程中，将一个函数地址赋给它，它就指向那个函数。函数指针变量赋值可按如下方式进行书写：

```
p=min;
```

可见在赋值时，只给出函数名称即可，不必给出函数的参数。

在使用函数指针变量调用函数时，要写出函数的参数，例如可按如下写法：

```
m=(*p)(a,b);
```

技巧 12　使用返回指针的函数查找最大值

在窗体上输入 10 个整数后，使用返回指针的函数实现在窗体上输出这些整数中的最大值。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）创建自定义函数 FindMax()实现返回输入的整数中的最大值。

（4）创建 main()函数，在此函数中调用指针函数 FindMax，将得到的最大值显示在窗体上。

例 10.12　创建自定义函数，使用主函数调用指针函数，将最大值显示出来。（光盘位置：光盘\MR\Instance\10\12）

```
#include<stdio.h>
*FindMax(int *p, int n)                    /*自定义函数*/
```

```
{
    int i,  *max;                                    /*定义整型变量和指针*/
    max = p;
    for (i = 0; i < n; i++)
        if (*(p + i) >  *max)
            max = p + i;                             /*把最大数的地址赋给变量*/
    return max;
}
void main()                                          /*main()主函数*/
{
    int a[10],  *max, i;                             /*定义数组、指针和变量*/
    printf("Please input ten integer:\n");           /*输出提示信息*/
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &a[i]);
    }
    max = FindMax(a, 10);
    printf("The max number is: %d\n",  *max);        /*输出查找到的最大值*/
    getch();
}
```

运行本实例，效果如图 10.16 所示。

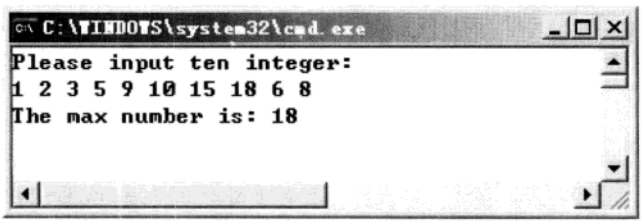


图 10.16　使用返回指针的函数查找最大值

技巧解析：

函数返回值可以是整型、字符型、实型等，同样也可以是指针型数值，即一个地址。

返回指针的函数的定义形式为：

```
int * fun(int x,int y)
```

在调用 fun 函数时，直接写函数名加上参数即可，返回一个指向整型数据的指针，其值为一个地址。x、y 是函数 fun 的形参。在函数名前面直接添加*，表示次函数是指针型函数，即函数值是指针。最前面的 int 表示返回的指针指向整型变量。

技巧 13　用指针函数实现求学生成绩

本技巧实现在窗体上输入学生序号，将在窗体上输出该序号对应的学生的成绩。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）创建自定义指针函数 search()实现按照学生序号进行查找。

（4）创建 main()函数，在此函数中调用指针函数 search，将得到的结果输出在窗体上。

例 10.13　自定义一个指针函数，在主函数中调用指针函数，求学生成绩。（光盘位置：光盘\MR\Instance\10\13）

```
#include<stdio.h>
float *search(float(*p)[4], int n)                          /*自定义指针函数*/
{
    float *pt;                                              /*定义指针*/
    pt = *(p + n);
    return (pt);                                            /*返回值*/
}
void main()                                                 /*main()主函数*/
{
    /*声明数组*/
    float score[][4]={{60,75,82,91},
                {75,81,91,90},
                {51,65,78,84},
                {65,51,78,72}};
    float *p;                                               /*定义指针*/
    int i, j;                                               /*定义整型变量*/
    printf("enter the student's number you want to search:");/*显示提示信息*/
    scanf("%d", &j);                                        /*输入学生学号*/
    printf("This student's scores are:\n");                 /*显示提示信息*/
    p = search(score, j);                                   /*调用指针函数*/
    for (i = 0; i < 4; i++)
        printf("%5.1f\t", *(p + i));
    getch();
}
```

运行本实例，效果如图 10.17 所示。

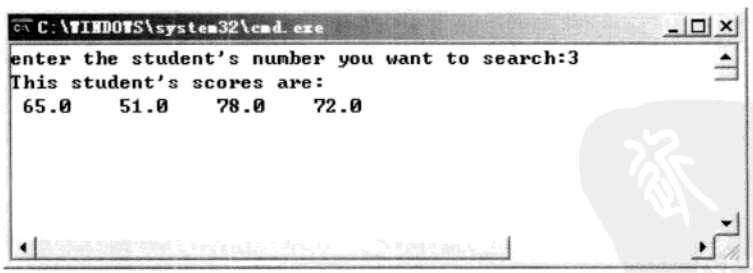


图 10.17　用指针函数实现求学生成绩

技巧 14　寻找指定元素的指针

在实际应用中，常常需要查找出数组中指定元素的指针地址，本技巧实现查找出所想要知道的数组中相应元素的指针。

例 10.14 创建两个自定义函数，一个函数实现查找指定值在数组中位置；另一个自定义函数实现返回指定值的指针，即返回指定值的地址。(光盘位置：光盘\MR\Instance\10\14)

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，声明数组。

```
#include <stdio.h>
int a[] =
{
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60
};
```

（3）创建自定义函数 search()，实现查找指定值在数组中位置；创建自定义函数 find()实现返回指定值的指针，即返回指定值的地址。

```
int search(int *pt, int n, int key)                    /*自定义函数 search()*/
{
    int *p;                                            /*定义指针*/
    for (p = pt; p < pt + n; p++)
      if (*p == key)                                   /*如果指针指向的值等于指定值，返回在数组中位置*/
          return p - pt;
    return 0;
}
int *find(int *pt, int n, int key)                     /*自定义函数 find()*/
{
    int *p;
    for (p = pt; p < pt + n; p++)                      /*如果指针指向的值等于指定值，返回地址*/
        if (*p == key)
            return p;
    return 0;
}
```

（4）创建 main()函数，在此函数中调用 search()函数和 find()函数，将得到的结果输出在窗体上。实现代码如下：

```
main()
{
    int i, key;                                                          /*定义整型变量*/
    int *j;                                                              /*定义指针*/
    printf("The elements of array a:\n");                                /*输出提示信息*/
    for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)                       /*输入数组 a 的元素*/
        printf("%d ", a[i]);
    printf("\nThe address of a[0] is :%d\n", &a[0]);
    printf("\nPlease input the key number you want to search:");        /*输入要查找的值*/
    i = search(a, sizeof(a) / sizeof(a[0]), key);                        /*调用自定义函数 search()*/
    scanf("%d", &key);
    /*输出查找值在数组中位置*/
    printf("\nThe label number of the key number %d in the array is: %d", key, i);
    j = find(a, sizeof(a) / sizeof(a[0]), key);                          /*调用自定义函数 find()*/
    /*输出查找值所在位置*/
    printf("\nThe point value of the key number %d in the array is: %d", key, j);
    getch();
}
```

运行本实例，效果如图 10.18 所示。

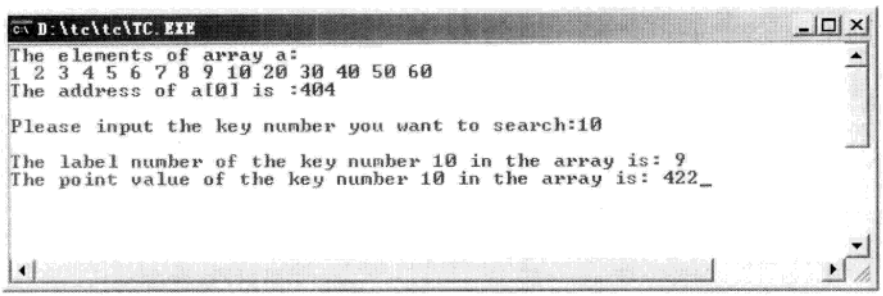


图 10.18　寻找指定元素的指针

技巧 15　寻找相同元素的指针

本技巧实现比较两个有序数组中的元素，输出两个数组中第一个相同的元素值。

例 10.15　比较两个有序数组中的数组元素，找出其中第一个相同的数组元素值，并将其显示在屏幕上。（光盘位置：光盘\MR\Instance\10\15）

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>
```

（3）创建自定义函数 find()，实现查找两个数组中第一个相同的元素值。

```
int *find(int *pa,int *pb,int an,int bn)                    /*自定义函数 find()*/
{
    int *pta,*ptb;                                          /*定义两个指针*/
    pta=pa;ptb=pb;
    while(pta<pa+an&&ptb<pb+bn)                             /*while 语句*/
    {
        if(*pta<*ptb)                                       /*如果指针*pta 小于*ptb*/
           pta++;
        else if(*pta>*ptb)                                  /*如果指针*pta 小于*ptb*/
           ptb++;
        else
           return pta;                                      /*如果两个值相等，返回这个值的指针*/
    }
    return 0;
}
```

（4）创建 main()函数，在此函数中调用 find()函数，将得到的结果输出在窗体上。实现代码如下：

```
void main()                                                 /*mian()主函数*/
{
    int *p, i;                                              /*定义指针和整型变量*/
    int a[] =                                               /*声明数组*/
    {
        1, 3, 5, 7, 9, 11, 13, 15                           /*数组 a 中的元素值*/
    };
```

```
    int b[] =                                                    /*声明数组*/
    {
        2, 4, 6, 8, 11, 15, 17                                   /*数组 b 中的元素值*/
    };
    printf("The elements of array a:");                          /*输出提示信息*/
    for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
        printf("%d ", a[i]);                                     /*输出数组 a 的元素*/
    printf("\nThe elements of array b:");                        /*输出提示信息*/
    for (i = 0; i < sizeof(b) / sizeof(b[0]); i++)
        printf("%d ", b[i]);                                     /*输出数组 b 的元素*/
    p=find(a,b,sizeof(a)/sizeof(a[0]),sizeof(b)/sizeof(b[0]));
    if (p)
        printf("\nThe first element in both arrays is %d\n",*p);
    else
        printf("Doesn't found the same element!\n");
    getch();
}
```

运行本实例，效果如图 10.19 所示。

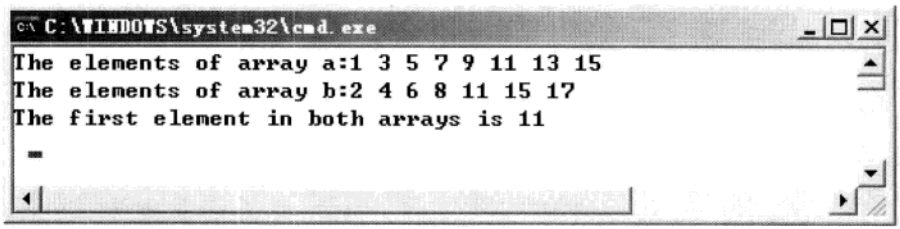


图 10.19　寻找相同元素的指针

技巧 16　使用指针的指针输出字符串

本技巧实现使用指针的指针输出字符串。首先要使用指针数组创建一个字符串数组，然后定义指向指针的指针，使其指向字符串数组，并使用其将数组中字符串输出。

例 10.16　使用指针数组创建字符串数组，并使用指向指针的指针将字符串数组中的字符串输出。（光盘位置：光盘\MR\Instance\10\16）

（1）在 TC 下创建一个 C 文件。

（2）创建 main()函数，在此函数中使用指针数组创建字符串数组，并使用指向指针的指针将字符串数组中的字符串输出。实现代码如下：

```
#include<stdio.h>
main()
{
    char *strings[]={"C language",
                     "Basic",
                     "World wide",
                     "Olympic",
                     "Great Wall"};                  /*使用指针数组创建字符串数组*/
    char **p,i;                                      /*声明变量*/
    p=strings;                                       /*指针指向字符串数组首地址*/
```

```
    for(i=0;i<5;i++)                                    /*循环输出字符串*/
    {
        printf("%s\n",*(p+i));
    }
}
```

运行本实例，效果如图 10.20 所示。

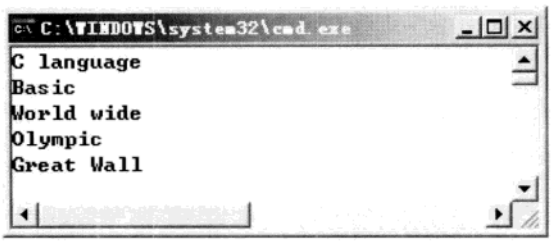


图 10.20　使用指针的指针输出字符串

技巧解析：

本实例使用指针的指针实现对字符串数组中字符串的输出。指向指针的指针即是指向指针数据的指针变量。这里创建一个指针数组 strings，它的每个数组元素相当于一个指针变量，都可以指向一个整型变量，其值为地址，如图 10.21 所示。strings 是一个数组，它的每个元素都有相应的地址。数组名 stirngs 代表该指针数组的首单元的指针，就是说指针数组首单元中存放的也是一个指针。strings+i 是 strings[i]的地址。strings+i 就是指向指针型数据的指针。

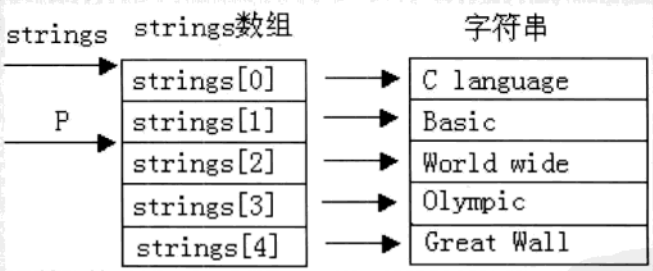


图 10.21　指针数组结构示意图

指向指针数据的指针变量定义语句形式如下：

```
char **P;
```

p 的前面有两个*号、*运算符是从右到左结合，**p 就相当于*（*p），*p 表示定义一个指针变量，在其前面再添加一个*号，表示指针变量 p 是指向一个指针变量。*p 就表示 p 所指向的另一个指针变量，即一个地址。**p 是 p 间接指向的对象的值。例如，这里*（p+2）就表示 strings[2]中的内容，它也是一个指针，指向字符串“World wide”。因此，输出字符串时，语句为：

```
printf("%s\n",*(p+i));
```

技巧 17 实现输入月份输出该月份英文名

本技巧使用指针数组创建一个含有月份英文名的字符串数组，并使用指向指针的指针指向这个字符串数组，实现输出数组中的制定字符串。运行程序后，输入要显示英文名的月份，将输出该月份对应的英文名。

实现思想：

与上个技巧一样，本技巧使用指针的指针实现对字符串数组中字符串的输出。这里首先定义一个包含月份英文名的字符串数组，并定义了一个指向指针的指针变量指向该数组。使用该变量输出字符串数组的字符串。

例 10.17 使用指针数组创建字符串数组，并使用指向指针的指针将字符串数组中的字符串输出。（光盘位置：光盘\MR\Instance\10\17）

（1）在 TC 下创建一个 C 文件。

（2）创建 main()函数，在此函数中使用指针数组创建字符串数组，并使用指向指针的指针将字符串数组中的字符串输出。实现代码如下：

```
#include "stdio.h"
int main()
{
    char *Month[]={                                  /*定义字符串数组*/
                   "January",
                   "February",
                   "March",
                   "April",
                   "May",
                   "June",
                   "Junly",
                   "August",
                   "September",
                   "October",
                   "November",
                   "December"
                   };
    int i;
    char **p;                                        /*声明指向指针的指针变量*/
    p=Month;                                         /*将数组首地址值赋给指针变量*/
    printf("Input a number for month\n");
    scanf("%d",&i);                                  /*输入要显示的月份号*/
    printf("The month is:");
    printf("%s\n",*(p+i-1));                         /*使用指向指针的指针输出对应的字符串数组中字符串*/
    getch();
    return 0;
}
```

运行本实例，效果如图 10.22 所示。

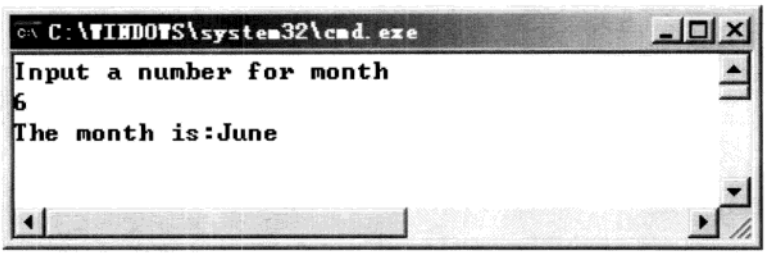


图 10.22　实现输入月份输出该月份英文名

技巧 18　使用指向指针的指针对字符串排序

排序的方法有很多种，例如冒泡法排序法、选择排序法、交换排序法和折半排序法等，本技巧是用指向指针的指针实现对字符串数组中的字符串排序输出。

实现思路：

本实例同样使用指向指针的指针实现对字符串数组中的字符串排序，这里定义了自定义函数 sort()，使用函数 strcmp()实现对给定字符串的比较，并进行排序。

例 10.18　创建自定义函数 sort()实现对字符串排序。创建主函数 main()函数，在此函数中调用 sort()函数对字符串数组中字符串排序，并将排序结果显示在窗体上。（光盘位置：光盘\MR\Instance\10\18）

（1）在 TC 下创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>                                    /*引用头文件*/
#include<string.h>
```

（3）创建自定义函数 sort()实现对字符串排序。

```
sort(char *strings[], int n)                         /*创建自定义函数 sort()*/
{
    char *temp;                                      /*声明字符型指针变量*/
    int i, j;                                        /*声明整型变量*/
    for (i = 0; i < n; i++)                          /*外层循环*/
    {
        for (j = i + 1; j < n; j++)
        {
            if (strcmp(strings[i], strings[j]) > 0)  /*比较两个字符*/
            {
                temp = strings[i];                   /*交换字符位置*/
                strings[i] = strings[j];
                strings[j] = temp;
            }
        }
    }
}
```

（4）创建 main()函数，在此函数中调用 sort()函数对字符串数组中字符串排序，并将排序结果显示在窗体上。实现代码如下：

```
void main()
{
    int n = 5;                                                    /*定义变量并初始化*/
    int i;                                                        /*定义整型变量*/
    char **p;                                                     /*指向指针的指针变量*/
    char *strings[] =
    {
        "C language", "Basic", "World wide", "Hello world", "Great Wall"
    };                                                            /*初始化字符串数组*/
    p = strings;                                                  /*指针指向数组首地址*/
    sort(p, n);                                                   /*调用排序自定义过程*/
    for (i = 0; i < n; i++)                                       /*循环输出排序后的数组元素*/
        printf("%s\n", strings[i]);
    getch();
}
```

运行本实例，效果如图 10.23 所示。

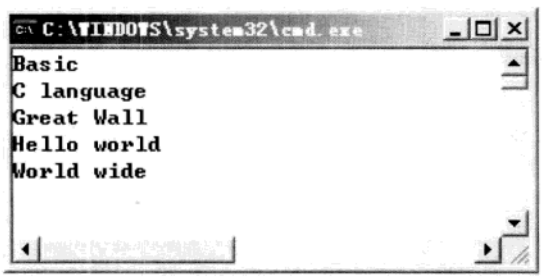


图 10.23　随机抽取双色球号码



第11章 文件开发技巧

在 C 语言中“文件”是一个逻辑概念，可以用来表示从磁盘文件到终端等所有东西，磁盘文件的操作有打开、读和写，而且磁盘文件有不同的读写方式，可以根据需要进行选择。在系统中可以对文件进行创建、查找和删除，同样在程序中也可以实现这样的功能。

在实际应用中经常会有一些文件是不希望被别人看到的，这时需要对文件进行加密，在文件操作中还可以对大文件进行分割、显示目录内同类型文件等。本章将逐一进行介绍。

技巧 01 读取磁盘文件

在磁盘上新建一个文本文档，文档内容为 hello world hello mingri!，从键盘中输入文件路径及名称，在屏幕中显示出该文件内容。要实现这样的功能可以通过 fopen 函数、fgetc 函数和 fclose 函数实现。

（1）fopen 是文件的打开函数。语法如下：

```
fp=fopen(文件名，使用文件方式)
```

- 文件名：指定磁盘文件的路径。
- 返回值：返回 FILE 类型的指针。
- 文件使用方式如表 11.1 所示。

表 11.1 文件使用方式及含义

文件使用方式	含　义
“r”（只读）	打开一个文本文件，只允许读数据
“w”（只写）	打开或建立一个文本文件，只允许写数据
“a”（追加）	打开一个文本文件，并在文件末尾写数据
“rb”（只读）	打开一个二进制文件，只允许读数据
“wb”（只写）	打开或建立一个二进制文件，只允许写数据
“ab”（追加）	打开一个二进制文件，并在文件末尾写数据
“r+”（读写）	打开一个文本文件，允许读和写
“w+”（读写）	打开或建立一个文本文件，允许读写

续表

文件使用方式	含　义
“a+”（读写）	打开一个文本文件，允许读，或在文件末追加数据
“rb+”（读写）	打开一个二进制文件，允许读和写
“wb+”（读写）	打开或建立一个二进制文件，允许读和写
“ab+”（读写）	打开一个二进制文件，允许读，或在文件末追加数据

fopen 函数的简单应用如下：

```
fp=fopen("123.txt","r");
```

它表示要打开名称为 123 的文本文档，使用文件方式为“只读”，fopen 函数带回指向 123.txt 文件的指针并赋给 fp，也就是说 fp 指向 123.txt 文件。

（2）fclose 是文件的关闭函数，通过文件指针将该文件关闭。语法如下：

```
fclose(文件指针)
```

- 文件指针：FILE 文件类指针。

（3）fgetc 函数是从指定的文件（fp 指向的文件）读入一个字符赋给 ch。

```
ch=fgetc(fp);
```

ch 是一个字符变量，fp 是一个 FILE 文件指针。

注意：fgetc 函数所打开的文件必须是以读或读写方式打开。

例 11.1　读取磁盘文件（光盘位置：光盘\MR\Instance\11\01\01.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    FILE *fp;                              /*定义一个指向 FILE 类型结构体的指针变量*/
    char ch, filename[50];                 /*定义变量及数组为字符型*/
    printf("please input file`s name:\n");
    gets(filename);                        /*输入文件所在路径及名称*/
    fp = fopen(filename, "r");             /*以只读方式打开指定文件*/
    ch = fgetc(fp);                        /*fgetc 函数带回一个字符赋给 ch*/
    while (ch != EOF)                      /*当读入的字符值等于 EOF 时结束循环*/
    {
        putchar(ch);                       /*将读入的字符输出在屏幕上*/
        ch = fgetc(fp);                    /*fgetc 函数继续带回一个字符赋给 ch*/
    }
    fclose(fp);                            /*关闭文件*/
}
```

程序运行结果如图 11.1 所示。

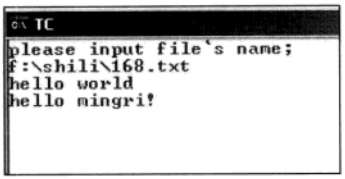


图 11.1　读取磁盘文件

技巧 02　将数据写入磁盘文件

首先在任意路径下新建一个文本文档，然后向该文档中写入字符串 mingri tu shu,nin shen bian de liang shi yi you!，在程序中通过字符“#”结束字符串的输入。实现数据向磁盘文件中写入可以使用 fputc 函数。

fputc 函数实现把一个字符写到磁盘文件。简单实例如下：

```
ch=fputc(ch,fp);
```

fputc 把一个字符写到 fp 所指向的文件中。其中 ch 是要输出的字符，它可以是一个字符常量，也可以是一个字符变量。fp 是文件指针变量。

例 11.2　将数据写入磁盘文件（光盘位置：光盘\MR\Instance\11\02\02.c），该实例的主要代码如下：

```
main()
{
    FILE *fp;                                   /*定义一个指向 FILE 类型结构体的指针变量*/
    char ch, filename[50];                      /*定义变量及数组为字符型*/
    printf("please input filename:\n");
    scanf("%s", filename);                      /*输入文件所在路径及名称*/
    if ((fp = fopen(filename, "w")) == NULL)    /*以只写方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    }
    ch = getchar();                             /*fgetc 函数带回一个字符赋给 ch*/
    while (ch != '#')                           /*当输入"#"时结束循环*/
    {
        fputc(ch, fp);                          /*将读入的字符写到磁盘文件上去*/
        ch = getchar();                         /*fgetc 函数继续带回一个字符赋给 ch*/
    }
    fclose(fp);                                 /*关闭文件*/
}
```

程序运行结果如图 11.2 所示。

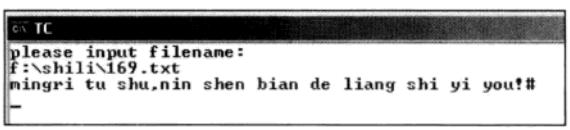


图 11.2　将数据写入磁盘文件

注意：程序需要用户文件名“169.txt”和一个字符串。

在程序中指定的文件名为 169.txt，通过程序写入数据后，效果如图 11.3 所示。

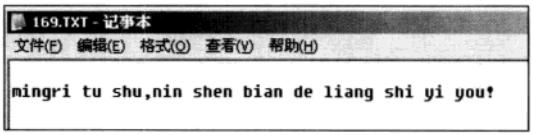


图 11.3 写入磁盘中的内容

技巧 03 格式化读写文件

在读取和写入文件时需要对数据进行规范化，将数据整齐地写入文件或读取出来。首先将输入的小写字母串写入磁盘文件，再将刚写入磁盘文件的内容读出并以大写字母的形式显示在屏幕上，读取字符时，需要借助 while 循环，用 while 循环来实现字符串的读入，通过设置的标志位来判断是否结束字符串读入。当 flag 为 1 时继续读入字符串，当 flag 为 0 时停止读入字符串。将读入的小写字母转换为大写字母，只需将读入的小写字母的 ASCII 减 32 即可实现，格式化读写文件需要通过 fprintf 函数和 fscanf 函数实现。

（1）fprintf 函数实现将指定格式的数据写入到文件。语法如下：

```
ch=fprintf(文件指针，格式字符串，输出列表);
```

fprintf 函数的简单应用如下：

```
fprintf(fp, "%d",i);
```

上面代码是将整型变量 i 的值按%d 的格式输出到 fp 指定的文件上。

（2）fscanf 函数实现将用户输入数据写入到文件，语法如下：

```
fscanf(文件指针,格式字符串,输入列表)
```

fprintf 函数的简单应用如下：

```
fscanf(fp,"%d",&i);
```

上面代码用于读入 fp 所指向的文件上的 i 的值。

注意：fprintf 和 fscanf 函数的读写对象不是终端而是磁盘文件。

例 11.3 格式化读写文件（光盘位置：光盘\MR\Instance\11\03\03.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
```

```
    int i, flag = 1;                                    /*定义变量为基本整型*/
    char str[80], filename[50];                         /*定义数组为字符型*/
    FILE *fp;                                           /*定义一个指向 FILE 类型结构体的指针变量*/
    printf("please input filename:\n");
    scanf("%s", filename);                              /*输入文件所在路径及名称*/
    if ((fp = fopen(filename, "w")) == NULL)            /*以只写方式打开指定文件*/
    {
        printf("cannot open!");
        exit(0);
    }
    while (flag == 1)
    {
        printf("\nInput string:\n");
        scanf("%s", str);                               /*输入字符串*/
        fprintf(fp, "%s", str);                         /*将 str 字符串内容以%s 形式写到 fp 所指文件上*/
        printf("\nContinue:?");
        if ((getchar() == 'N') || (getchar() == 'n')) /*输入 n 结束输入*/
            flag = 0;                                   /*标志位置 0*/
    }
    fclose(fp);                                         /*关闭文件*/
    fp = fopen(filename, "r");                          /*以只写读方式打开指定文件*/
    while (fscanf(fp, "%s", str) != EOF)                /*从 fp 所指的文件中以%s 形式读入字符串*/
    {
        for (i = 0; str[i] != '\0'; i++)
            if ((str[i] >= 'a') && (str[i] <= 'z'))
                str[i] -= 32;                           /*将小写字
母转换为小写字母*/
        printf("\n%s\n", str);                          /*输出转换
后的字符串*/
    }
    fclose(fp);                                         /*关闭文件
*/
}
```

```
TC
please input filename:
f:\shili\170.txt

Input string:
helloworld

Continue:?n

HELLOWORLD
```

图 11.4　巧用位移运算符获得汉字编码

程序运行结果如图 11.4 所示。

技巧 04　成块读写操作

当文件数据量比较大时，不能一个字符一个字符的对文件进行读取，需要一次就读取和写入多个数据。通过 fwrite 函数及 fread 函数可以实现学生成绩信息统计，从键盘中输入学生成绩信息，保存到指定磁盘文件中，输入完全部信息后将磁盘文件中保存的信息输出到屏幕上。

（1）fwrite 函数。

该函数将 buffer 地址开始的信息输出 count 次，每次写 size 字节到 fp 所指的文件中。

```
fwrite(buffer,size,count,fp);
```

参数说明：

- buffer：是一个指针，是要输出数据的地址（起始地址）。
- size：要读写的字节数。
- count：要进行读写多少个 size 字节的数据项。

- fp：文件型指针。

（2）fread 函数。

该函数是从 fp 所指的文件中读入 count 次，每次读 size 字节，读入的信息存在 buffer 地址中。

```
fread(buffer,size,count,fp)
```

参数说明同 fwrite 函数，唯一不同是 buffer 是读入数据的存放地址。

例 11.4 成块读写操作（光盘位置：光盘\MR\Instance\11\04\04.c），该实例的主要代码如下：

```
#include <stdio.h>
struct student_score                                    /*定义结构体存储学生成绩信息*/
{
    char name[10];
    int num;
    int China;
    int Math;
    int English;
} score[100];
//自定义 save ( ) 函数，作用是将输入的一组数据输出到指定的磁盘文件中去。
void save(char *name, int n)                            /*自定义函数 save*/
{
    FILE *fp;                                           /*定义一个指向 FILE 类型结构体的指针变量*/
    int i;
    if ((fp = fopen(name, "wb")) == NULL)               /*以只写方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    }
    for (i = 0; i < n; i++)
        if (fwrite(&score[i], sizeof(struct student_score), 1, fp) != 1)/*将一组数据输出到 fp 所指的文件
中*/
            printf("file write error\n");               /*如果写入文件不成功，则输出错误*/
    fclose(fp);                                         /*关闭文件*/
}
//自定义 show ( ) 函数，作用是将从指定的文件中读入的一组输出显示到屏幕上。
void show(char *name, int n)                            /*自定义函数 show*/
{
    int i;
    FILE *fp;                                           /*定义一个指向 FILE 类型结构体的指针变量*/
    if ((fp = fopen(name, "rb")) == NULL)               /*以只读方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    } for (i = 0; i < n; i++)
{
        /*从 fp 所指向的文件读入数据存到数组 score 中*/
        fread(&score[i], sizeof(struct student_score), 1, fp);
        printf("%-10s%4d%4d%4d%4d\n", score[i].name, score[i].num,
           score[i].China, score[i].Math, score[i].English);
    }
    fclose(fp);                                         /*以只写方式打开指定文件*/
}
//main()函数作为程序的入口函数。
main()
{
    int i, n;                                           /*变量类型为基本整型*/
```

```
    char filename[50];                                          /*数组为字符型*/
    printf("how many students in your class?\n");
    scanf("%d", &n);                                            /*输入学生数*/
    printf("please input filename:\n");
    scanf("%s", filename);                                      /*输入文件所在路径及名称*/
    printf("please input name,number,China,math,English:\n");
    for (i = 0; i < n; i++)                                     /*输入学生成绩信息*/
    {
        printf("NO%d", i + 1);
        scanf("%s%d%d%d%d", score[i].name, &score[i].num, &score[i].China,
            &score[i].Math, &score[i].English);
        save(filename, n);                                      /*调用函数 save*/
    } show(filename, n);                                        /*调用函数 show*/
}
```

程序运行结果如图 11.5 所示。

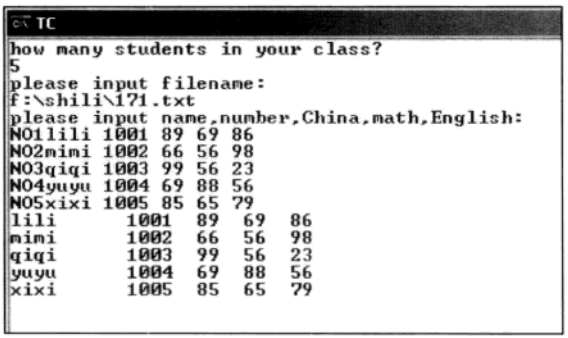


图 11.5　成块读写操作

技巧 05　随机读写文件

在读取和写入文件数据时，有时需要从文件的中间部分开始，这就需要通过移动文件指针来实现。输入若干个学生信息，保存到指定磁盘文件中，可以通过 fseek 函数将奇数条学生信息从磁盘中读入并显示在屏幕上。

fseek 函数用来移动文件内部位置指针。其中参数“起始点”表示从何处开始计算位移量，规定的起始点有 3 种：文件首、文件当前位置和文件尾。

```
fseek(文件类型指针,位移量,起始点);
```

- 文件类型指针：FILE 类型的文件指针。
- 位移量：从参数“起始点”开始移动多远的距离。
- 起始点：表示从何处开始计算位移量，规定的起始点有 3 种：文件首（SEEK—SET）、文件当前位置（SEEK—CUR）和文件尾（SEEK—END）。

fseek 函数的简单应用如下：

```
fseek(fp,-20L,1);
```

上面代码表示将位置指针从当前位置向后退 20 个字节。

例 11.5 随机读写文件（光盘位置：光盘\MR\Instance\11\05\05.c），该实例的主要代码如下：

```
#include <stdio.h>
struct student_score                                    /*定义结构体存储学生成绩信息*/
{
    char name[10];
    int num;
    int China;
    int Math;
    int English;
} score[100];
//自定义 save()函数，作用是将输入的一组数据输出到指定的磁盘文件中去
void save(char *name, int n)                            /*自定义函数 save*/
{
    FILE *fp;                                           /*定义一个指向 FILE 类型结构体的指针变量*/
    int i;
    if ((fp = fopen(name, "wb")) == NULL)               /*以只写方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    }
for (i = 0; i < n; i++)
        /*将一组数据输出到 fp 所指的文件中*/
        if (fwrite(&score[i], sizeof(struct student_score), 1, fp) != 1)
           printf("file write error\n");                /*如果写入文件不成功，则输出错误*/
    fclose(fp);                                         /*关闭文件*/
}
//自定义 show ( ) 函数，作用是将从指定的文件中读入的一组输出显示到屏幕上。
void show(char *name, int n)                            /*自定义函数 show*/
{
    int i;
    FILE *fp;                                           /*定义一个指向 FILE 类型结构体的指针变量*/
    if ((fp = fopen(name, "rb")) == NULL)               /*以只读方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(0);
    } for (i = 0; i < n; i++)
{
        /*从 fp 所指向的文件读入数据存到数组 score 中*/
        fread(&score[i], sizeof(struct student_score), 1, fp);
        printf("%-10s%4d%4d%4d%4d\n", score[i].name, score[i].num,
           score[i].China, score[i].Math, score[i].English);
    }
    fclose(fp);                                         /*以只写方式打开指定文件*/
}
//main()函数作为程序的入口函数
main()
{
    int i, n;                                           /*变量类型为基本整型*/
    char filename[50];                                  /*数组为字符型*/
    printf("how many students in your class?\n");
    scanf("%d", &n);                                    /*输入学生数*/
    printf("please input filename:\n");
    scanf("%s", filename);                              /*输入文件所在路径及名称*/
    printf("please input name,number,China,math,English:\n");
    for (i = 0; i < n; i++)                             /*输入学生成绩信息*/
    {
```

```
        printf("NO%d", i + 1);
        scanf("%s%d%d%d%d", score[i].name, &score[i].num, &score[i].China,
            &score[i].Math, &score[i].English);
        save(filename, n);                                    /*调用函数 save*/
    } show(filename, n);                                      /*调用函数 show*/
}
```

程序运行结果如图 11.6 所示。

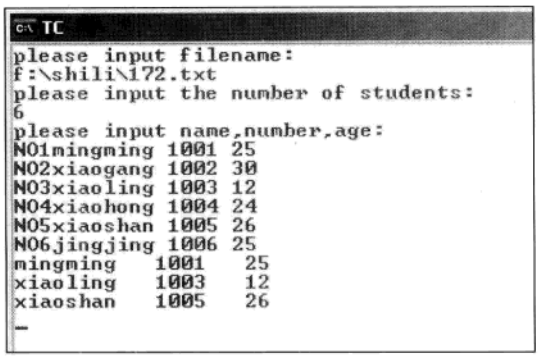


图 11.6　随机读写文件

技巧 06　复制文件内容到另一文件

实现文件复制的过程并不是很复杂，将一个已存在的文本文档的内容复制到新建的文本文档中，可以使用 while 循环从被复制的文件中逐个读取字符到另一个文件中。

注意：在实现复制的过程中无论是复制的文件还是被复制的文件都应该是打开的状态，当复制完成后再将两个文件分别关闭。

例 11.6　复制文件内容到另一文件（光盘位置：光盘\MR\Instance\11\06\06.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    FILE *in,*out;                                    /*定义两个指向 FILE 类型结构体的指针变量*/
    char ch, infile[50], outfile[50];                 /*定义数组及变量为基本整型*/
    printf("Enter the infile name:\n");
    scanf("%s", infile);                              /*输入将要被复制的文件所在路径及名称*/
    printf("Enter the outfile name:\n");
    scanf("%s", outfile);                             /*输入新建的将用于复制的文件所在路径及名称*/
    if ((in = fopen(infile, "r")) == NULL)            /*以只写方式打开指定文件*/
    {
        printf("cannot open infile\n");
        exit(0);
    }
    if ((out = fopen(outfile, "w")) == NULL)
    {
        printf("cannot open outfile\n");
```

```
        exit(0);
    }
    ch = fgetc(in);
    while (ch != EOF)
    {
        fputc(ch, out);                    /*将 in 指向的文件的内容复制到 out 所指向的文件中*/
        ch = fgetc(in);
    }
    fclose(in);
    fclose(out);
}
```

程序运行结果如图 11.7 所示。

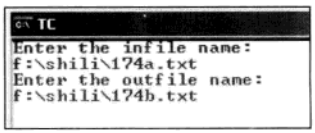


图 11.7　复制文件内容到另一文件

程序运行需要用户输入两个文件名，一个为“174A”，另一个为“174B”，文件“174A”的内容如图 11.8 所示，文件“174B”的内容如图 11.9 所示。

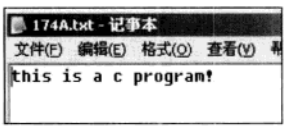


图 11.8　174A 内容

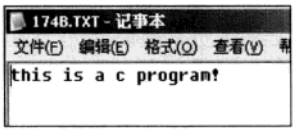


图 11.9　174B 内容

技巧 07　合并两个文件信息

首先建立两个文本文件，第一个文本文件的内容是：“hello computer!!”，第二个文本文件的内容是：“This is a c program!!”，将文件二的内容合并到文件一内容的后面。

首先将文件二中的内容逐个读取并显示到屏幕上，当将文件二中的全部内容读取后位置指针 fp2 也就指到了文件末尾处，此时需要将文件二中的位置指针 fp2 重新移到文件开始处，然后将文件二中的内容逐个合并（文件二读取操作，文件一写入操作）到文件 1 中。

通过 3 次 while 循环实现文件的合并，前两次使用 while 循环是为了将两文件中原有内容显示在屏幕上，第三次使用 while 循环目的是将文件二中的内容逐个写入文件一中，从而实现合并。

例 11.7　合并两个文件信息（光盘位置：光盘\MR\Instance\11\07\07.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    char ch, filename1[50], filename2[50];          /*数组和变量的数据类型为字符型*/
    FILE *fp1,  *fp2;                               /*定义两个指向 FILE 类型结构体的指针变量*/
    printf("please input filename1:\n");
    scanf("%s", filename1);                         /*输入文件所在路径及名称*/
    if ((fp1 = fopen(filename1, "a+")) == NULL)     /*以读写方式打开指定文件*/
    {
        printf(" cannot open\n");
        exit(0);
```

```
    }
    printf("file1:\n");
    ch = fgetc(fp1);
    while (ch != EOF)
    {
        putchar(ch);                                    /*将文件 1 中的内容输出*/
        ch = fgetc(fp1);
    }
    printf("\nplease input filename2:\n");
    scanf("%s", filename2);                             /*输入文件所在路径及名称*/
    if ((fp2 = fopen(filename2, "r")) == NULL)          /*以只读方式打开指定文件*/
    {
        printf("cannot open\n");
        exit(0);
    }
    printf("file2:\n");
    ch = fgetc(fp2);
    while (ch != EOF)
    {
        putchar(ch);                                    /*将文件 2 中的内容输出*/
        ch = fgetc(fp2);
    }
    fseek(fp2, 0L, 0);                                  /*将文件 2 中的位置指针移到文件开始处*/
    ch = fgetc(fp2);
    while (!feof(fp2))
    {
        fputc(ch, fp1);                                 /*将文件 2 中的内容输出到文件 1 中*/
        ch = fgetc(fp2);                                /*继续读取文件 2 中的内容*/
    }
    fclose(fp1);                                        /*关闭文件 1*/
    fclose(fp2);                                        /*关闭文件 2*/
}
```

程序运行结果如图 11.10 所示。

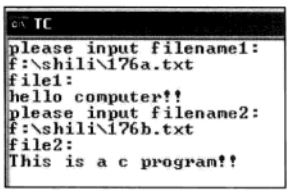


图 11.10　合并两个文件信息

程序运行需要用户输入两个文件名，一个为“176a”，另一个为“176b”，文件“176a”的内容如图 11.11 所示，合并后的内容如图 11.12 所示。

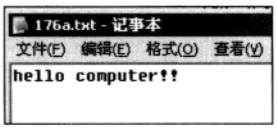


图 11.11　176a 内容

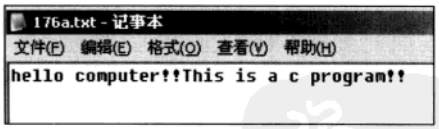


图 11.12　176b 内容

技巧 08　文件分割

文件分割就是将一个文件中的内容分割到多个文件内，分割时要注意扩展名问题和文件大小问题：要生成新的文件就要给新的文件命名，命名的过程中要注意不能重复命名，另外新文件要与原文件的扩展名一致。文件大小可以通过 ftell 和 fseek 函数组合来获得，通过 ftell 和 fseek 函数可以获取源文件大小，也可以控制分割后文件大小。

例 11.8　文件分割（光盘位置：光盘\MR\Instance\11\08\08.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
FILE *in,  *out;                                    /*定义两个指向 FILE 类型结构体的指针变量*/
char filename[50], ch, cfilename[50];
//自定义 space()函数，用来实现文件的分割，代码如下：
void space()                                        /* 分隔文件函数 */
{
    char ext[6][6] =
    {
        "a.txt", "b.txt", "c.txt", "d.txt", "e.txt", "f.txt"
    };                                              /* 分割出来的文件扩展名 */
    unsigned long int n = 1, k, byte = 0;           /* 定义变量类型为无符号的长整型变量 */
    unsigned int j = 0, i = 0;
    printf("Please input filename:\n");
    scanf("%s", filename);                          /*输入文件所在路径及名称*/
    strcpy(cfilename, filename);                    /*输入文件所在路径及名称复制到 cfilename 中*/
    if ((in = fopen(filename, "r")) == NULL)        /* 以只读方式打开输入文件 */
    {
        printf("Cannot open file\n");
        exit(0);
    }
    printf("Please input file size after space(kb):\n");
    scanf("%d", &n);                                /* 输入分割后单个文件的大小 */
    n = n * 1024;
    while (filename[j] != '.')
        j++;
    filename[j] = '\0';                             /*遇'.'时,在该处加字符串结束符*/
    if ((out = fopen(strcat(filename, ext[i]), "w")) == NULL)   /*生成分割后文件所在路径及名称*/
    {
        printf("Cannot open file\n");
        exit(0);
    }
    fseek(in, 0, 2);                                /*将位置指针移到文件末尾*/
    k = ftell(in);                                  /* k 存放当前位置，也就是整个文件的大小 */
    fseek(in, 0, 0);
    while (k > 0)
    {
        ch = fgetc(in);
        fputc(ch, out);
        byte++;                                     /*字节数增加*/
        k--;                                        /*大小减 1*/
        if (byte == n)                              /*当为要求的大小时执行括号内语句*/
        {
            fclose(out);                            /*完成一个分割出的文件*/
            byte = 0;                               /*byte 重新置 0*/
            strcpy(filename, cfilename);            /*filename 恢复初始状态*/
            while (filename[j] != '.')
                j++;
            filename[j] = '\0';                     /*遇'.'时,在该处加字符串结束符*/
            i++;
            if ((out = fopen(strcat(filename, ext[i]), "w")) == NULL)
                                                    /*生成分割后文件所在路径及名称*/
            {
                printf("Cannot open file\n");
                exit(0);
            }
        }
    }
    fclose(in);                                     /* 关闭文件 */
    printf("File succeed space!\n\n\n");
}
```

```
main()                                                    /* 程序主函数 */
{
    printf(" now file space!\n");
    space();
}
```

程序运行结果如图 11.13 所示。

程序运行时需要用户输入一个文件名，例如输入“177.TXT”，实例会将文件 177.txt 分割成 6 个文件，分割后如图 11.14 所示。

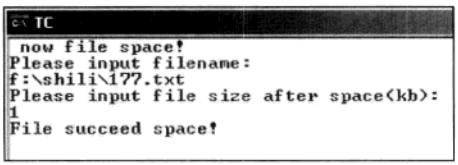


图 11.13　文件分割

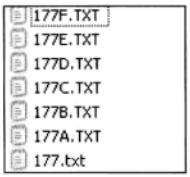


图 11.14　巧用位移运算符获得汉字编码

技巧 09　文件加密

对文本文档中的内容进行加密，实质上就是读取该文档中的内容，并对读出的每个字符与输入的密码进行异或，再将异或后的内容重新写入指定的磁盘文件中即可。先从键盘中输入要加密操作的文件所在的路径及名称，再输入密码，最后输入加密后的文件要存储路径及名称，这样就完成了一个加密过程。

例 11.9　文件加密（光盘位置：光盘\MR\Instance\11\09\09.c），该实例的主要代码如下：

```
#include <stdio.h>                                        /*标准输入输出头文件*/
#include <stdlib.h>
#include <string.h>
//自定义 encrypt ( ) 函数，作用是实现对指定文件进行加密
void encrypt(char *s_file, char *pwd, char *c_file)   /*自定义函数 encrypt 用于加密*/
{
    int i = 0;
    FILE *fp1,  *fp2;                                     /*定义 fp1 和 fp2 是指向结构体变量的指针*/
    register char ch;
    fp1 = fopen(s_file, "rb");
    if (fp1 == NULL)
    {
        printf("cannot open s_file.\n");
        exit(1);                                          /*如果不能打开要加密的文件，便退出程序*/
    }
    fp2 = fopen(c_file, "wb");
    if (fp2 == NULL)
    {
        printf("cannot open or create c_file.\n");
        exit(1);                                          /*如果不能建立加密后的文件，便退出*/
    }
    ch = fgetc(fp1);
    while (!feof(fp1))                                    /*测试文件是否结束*/
    {
        ch = ch ^ *(pwd + i);                             /*采用异或方法进行加密*/
```

```
            i++;
            fputc(ch, fp2);                          /*异或后写入 fp2 文件*/
            ch = fgetc(fp1);
            if (i > 9)
                i = 0;
        }
        fclose(fp1);                                 /*关闭源文件*/
        fclose(fp2);                                 /*关闭目标文件*/
    }
    //main()函数作为程序的入口函数
    main(int argc, char *argv[])                     /*定义 main()函数的命令行参数*/
    {
        char sourcefile[50];                         /*用户输入的要加密的文件名*/
        char codefile[50];
        char pwd[10];                                /*用来保存密码*/
        if (argc != 4)                               /*容错处理*/
        {
            printf("please input encode file name:\n");
            gets(sourcefile);                        /*得到要加密的文件名*/
            printf("please input Password:\n");
            gets(pwd);                               /*得到密码*/
            printf("please input saved file name:\n");
            gets(codefile);                          /*得到加密后你要的文件名*/
            encrypt(sourcefile, pwd, codefile);
        }
        else
        {
            strcpy(sourcefile, argv[1]);
            strcpy(pwd, argv[2]);
            strcpy(codefile, argv[3]);
            encrypt(sourcefile, pwd, codefile);
        }
    }
```

程序运行结果如图 11.15 所示。

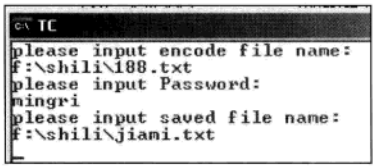


图 11.15　程序运行界面

程序运行需要用户输入一个加密前文件名和加密后文件名，加密前文件名为“188.txt”，文件中的内容如图 11.16 所示。加密后的文件名为“JIAMI.TXT”，文件内容如图 11.17 所示。

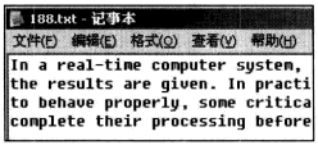


图 11.16　加密前文档中的部分内容

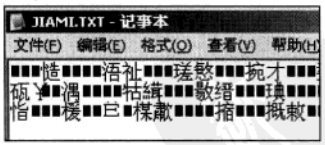


图 11.17　加密后文档中的内容

技巧 10　统计文件内容

统计输入字符串中字符的个数大家可能已经不再陌生，很多练习题中都会出现这种程序，可以对这种实例进行延伸，使其更有实用性。延伸后的功能主要是输入要进行统计的文件的路径及名称，统计出该文件中字符、空格、数字及其他字符的个数，并将统计结果存到指定的磁盘文件中。

统计功能的实现主要是靠条件判断实现的。在 while 循环中遍历要统计的文件中的每个字符，用条件判断语句对读入的字符进行判断并在相应的用于统计的变量（计数器）数上加 1，最后计数器的值就是所要求的结果。

例 11.10　统计文件内容（光盘位置：光盘\MR\Instance\11\10\10.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    FILE *fp1,  *fp2;                                   /*定义两个指向 FILE 类型结构体的指针变量*/
    char filename1[50], filename2[50], ch;              /*定义数组及变量为字符型*/
    long character, space, other, digit;                /*定义变量为长整型*/
    character = space = digit = other = 0;              /*长整形变量的初值均为 0*/
    printf("Enter file name \n");
    scanf("%s", filename1);                             /*输入要进行统计的文件的路径及名称*/
    if ((fp1 = fopen(filename1, "r")) == NULL)
     /*以只读方式打开指定文件*/
    {
        printf("cannot open file\n");
        exit(1);
    }
    printf("Enter file name for write data:\n");
    scanf("%s", filename2);                             /*输入文件名即将统计结果放到那个文件中*/
    if ((fp2 = fopen(filename2, "w")) == NULL)          /*以可写方式要存放统计结果的文件*/
    {
        printf("cannot open file\n");
        exit(1);
    }
    while ((ch = fgetc(fp1)) != EOF)                    /*知道文件内容结束处停止 while 循环*/
        if (ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z')
            character++;                                /*当遇到字母时字符个数加 1*/
    else if (ch == ' ')
        space++;                                        /*当遇到空格时空格数加 1*/
    else if (ch >= '0' && ch <= '9')
        digit++;                                        /*当遇到数字时数字数加 1*/
    else
        other++;                                        /*当是其他字符时其他字符数加 1*/
    close(fp1);                                         /*关闭 fp1 指向的文件*/
    fprintf(fp2, "character:%ld space:%ld digit:%ld other:%ld\n", character,
        space, digit, other);                           /*将统计结果写入 fp 指向的磁盘文件中*/
    fclose(fp2);                                        /*关闭 fp2 指向的文件*/
}
```

程序运行结果如图 11.18 所示。

程序运行时，需要用户输入目标文件，文件名为“185.txt”，还需要输入保存统计信息的文件名“JILU.TXT”，文件“JILU.TXT”的内容如图 11.19 所示。

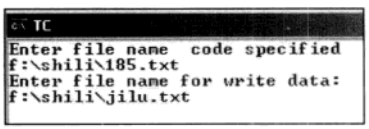


图 11.18　统计文件内容



图 11.19　统计后信息存在记事本中

技巧 11 创建文件

创建文件可以使用 creat 函数，首先从键盘中输入要创建的文件所在的路径及名称，然后根据创建结果输出提示信息。

create 函数生成一个新文件，如果函数执行成功，返回一个句柄给文件，如果出错，函数返回 -1，该函数在“io.h”头文件中。语法如下：

```
int creat(char *filename,int access);
```

参数说明：

- filename：任一有效文件名。
- access：用来指定访问的模式和标明该文件为二进制文件还是文本文件。一般情况下，生成一个标准存档文件时 access 的值为 0。access 取值及含义如表 11.2 所示。

表 11.2 access 取值及含义

位　号	值	含　义
0	1	只读文件
1	2	隐含文件
2	4	系统文件
3	8	卷标号名
4	16	子目录名
5	32	数据档案
6	64	未定义
7	128	未定义

例 11.11 创建文件（光盘位置：光盘\MR\Instance\11\11\11.c），该实例的主要代码如下：

```
#include<stdio.h>
#include<io.h>
main()
{
    int handle;
    char filename[50];                          /*数组定义为字符类型*/
    printf("please input filename:\n");
    scanf("%s", filename);                      /*输入要创建的文件所在路径及名称*/
    if ((handle = creatnew(filename, 0)) ==  - 1)
    {
        printf("\n Error! Cannot vreat!");      /*如果文件创建失败，输出双引号中内容*/
    }
    else
    {
        printf("\nthis file has created!");     /*如果文件创建成功，输出双引号中内容*/
```

```
        close(handle);
    }
}
```

程序运行结果如图 11.20 所示。

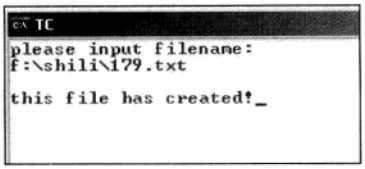


图 11.20　创建文件

技巧 12　创建临时文件

临时文件的创建可以通过 tmpfile 和 rewind 函数实现，可以实现将字符串“hello world hello mingri”输出到临时文件之后，再读取临时文件中内容，并将其显示在屏幕上。

（1）tmpfile 函数。

该函数实现创建一个临时文件。语法如下：

```
FILE *tmpfile()
```

- 返回值：如果函数执行成功，它以读和写模式打开文件，返回一个文件指针。如果出错，返回 NULL。

（2）rewind 函数。

该函数实现将文件的指针重新设置到该文件的起点。语法如下：

```
void rewind(FILE *fp)
```

- fp：FILE 类型文件指针。

例 11.12　创建临时文件（光盘位置：光盘\MR\Instance\11\12\12.c），该实例的主要代码如下：

```
#include<stdio.h>
main()
{
    FILE *temp;                                          /*定义一个指向 FILE 类型结构体的指针变量*/
    char c;                                              /*定义变量 c 为字符型*/
    if ((temp = tmpfile()) != NULL)
        fputs("\nhello world\nhello mingri", temp);      /*向临时文件中写入要求内容*/
    rewind(temp);                                        /*文件指针返回文件首*/
    while ((c = fgetc(temp)) != EOF)                     /*读取临时文件中内容*/
        printf("%c", c);                                 /*将读取的内容输出在屏幕上*/
    fclose(temp);                                        /*关闭临时文件*/
}
```

程序运行结果如图 11.21 所示。

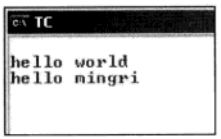


图 11.21　创建临时文件

技巧 13　重命名文件

使用 rename 函数可以实现重命名文件，从键盘中输入要重命名的文件的路径及名称，文件打开成功后输入新的路径及名称。

rename 函数的作用是把文件的名字从旧文件名改为新文件名。该函数的原型在“stdio.h”中。语法如下：

```
int rename(char *oldfname,char*newfname)
```

- oldfname：旧文件名。
- newfname：新文件名。
- 返回值：函数调用成功时返回 0，出错时返回非零值。

说明：oldfname 和 newfname 中的目录可以不同，因此可用 rename 把文件从一个目录移到另一个目录。

例 11.13　重命名文件（光盘位置：光盘\MR\Instance\11\13\13.c），该实例的主要代码如下：

```
#include<stdio.h>
main()
{
    FILE *fp;                                          /*定义一个指向 FILE 类型结构体的指针变量*/
    char filename1[20], filename2[20];                 /*定义数组为字符型*/
    printf("please input the file name which do you want to change:\n");
    scanf("%s", filename1);                            /*输入要重命名的文件所在的路径及名称*/
    if ((fp = fopen(filename1, "r")) != NULL)          /*以只读方式打开指定文件*/
    {
        printf("%s open successfully", filename1);
        printf("\nplease input new name!")??scanf("%s", filename2);   /*输入新的文件路径及名称*/
        rename(filename1, filename2);                  /*调用 rename 函数进行重命名*/
        fclose(fp);                                    /*关闭文件*/
    }
    else
    {
        printf("cannot find!!");
        exit(0);
    }
}
```

程序运行结果如图 11.22 所示。

```
TC
please input the file name which do you want to change:
f:\shili\182.txt
f:\shili\182.txt open successfully
please input new name!f:\shili\aaa.txt
```

图 11.22　重命名文件

技巧 14　删除文件

使用 remove 函数可以实现文件的删除，首先从键盘中输入要删除的文件的路径及名称，然后根据删除情况在屏幕中给出提示信息。

remove 函数的作用是删除 filename 所指定的文件。remove 函数的原型在“stdio.h”中。

```
int remove(char *filename)
```

- filename：设置所要删除文件的文件名。
- 返回值：删除成功返回 0，出现错误返回-1。

例 11.14　删除文件（光盘位置：光盘\MR\Instance\11\14\14.c），该实例的主要代码如下：

```
#include<stdio.h>
main()
{
    FILE *fp;                                   /*定义一个指向 FILE 类型结构体的指针变量*/
    char filename[50];                          /*定义数组为字符型*/
    printf("please input the name of the file which do you want to delete:\n");
    scanf("%s", filename);                      /*输入要删除的文件的路径及名称*/
    if ((fp = fopen(filename, "r")) != NULL)    /*以只读方式打开指定文件*/
    {
        printf("%s open successfully!", filename);/*文件打开成功输出提示信息*/
        fclose(fp);                             /*关闭文件*/
    }
    else
    {
        printf("%s cannot open!", filename);    /*文件打开失败输出提示信息*/
        exit(0);
    }
    remove(filename);                           /*调用函数删除文件*/
    if ((fp = fopen(filename, "r")) == NULL)
        printf("\n%s has removed!", filename);  /*若要打开的文件不存在，则删除成功，输出提示信息*/
    else
        printf("error");                        /*若要打开的文件存在，则删除不成功，输出提示信息*/
}
```

程序运行结果如图 11.23 所示。

```
TC
please input the name of the file which do you want to delete:
f:\shili\183.txt
f:\shili\183.txt open successfully!
f:\shili\183.txt has removed!_
```

图 11.23　删除文件

技巧 15 删除文件中的记录

要删除文件中的某条记录，需要先在文件中定位该条记录。下面以删除职工工资信息为例介绍删除的过程。首先打开一个二进制文件，此时打开应以追加的方式打开，要是以只写的方式打开会使文件中原有内容丢失，向该文件中输入员工工资信息，输入完毕将文件中内容全部输出；然后输入要删除的员工的姓名，使用 strcmp 函数查找相匹配的姓名来确定要删除记录的位置，将该位置后的记录分别前移一位，也就是将要删除的记录用后面的记录覆盖了；最后将删除后剩余的记录使用 fwrite 再次输出到磁盘文件中，使用 fread 函数读取文件内容到 emp 数组中并显示在屏幕上。

例 11.15 删除文件中的记录（光盘位置：光盘\MR\Instance\11\15\15.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <string.h>
struct emploee                                          /*定义结构体，存放员工工资信息*/
{
    char name[10];
    int salary;
} emp[20];
main()
{
    FILE *fp1,  *fp2;
    int i, j, n, flag, salary;
    char name[10], filename[50];                        /*定义数组为字符类型*/
    printf("please input filename:\n");
    scanf("%s", filename);                              /*输入文件所在路径及名称*/
    printf("please input the number of emploees:\n");
    scanf("%d", &n);                                    /*输入要录入的人数*/
    printf("input name and salary:\n");
    for (i = 0; i < n; i++)
    {
        printf("NO%d:\n", i + 1);
        scanf("%s%d", emp[i].name, &emp[i].salary);     /*输入员工姓名及工资*/
    }
    if ((fp1 = fopen(filename, "ab")) == NULL)          /*以追加的方式打开指定的二进制文件*/
    {
        printf("Can not open the file.");
        exit(0);
    }
for (i = 0; i < n; i++)
        /*将输入的员工信息输出到磁盘文件上*/
        if (fwrite(&emp[i], sizeof(struct emploee), 1, fp1) != 1)
           printf("error\n");
    fclose(fp1);
    if ((fp2 = fopen(filename, "rb")) == NULL)
    {
        printf("Can not open file.");
        exit(0);
} printf("\n original data:");
    /*读取磁盘文件上的信息到 emp 数组中*/
    for (i = 0; fread(&emp[i], sizeof(struct emploee), 1, fp2) != 0; i++)
        printf("\n %8s%7d", emp[i].name, emp[i].salary);
    n = i;
    fclose(fp2);
```

```
    printf("\n Input name which do you want to delete:");
    scanf("%s", name);                                          /*输入要删除的员工姓名*/
    for (flag = 1, i = 0; flag && i < n; i++)
    {
        if (strcmp(name, emp[i].name) == 0)                     /*查找与输入姓名相匹配的位置*/
        {
            for (j = i; j < n - 1; j++)
{
                /*查找到要删除信息的位置后将后面信息前移*/
                strcpy(emp[j].name, emp[j + 1].name);
                emp[j].salary = emp[j + 1].salary;
            } flag = 0;                                         /*标志位置 0*/
        }
    }
    if (!flag)
        n = n - 1;                                              /*记录个数减 1*/
    else
        printf("\nNot found");
    printf("\nNow,the content of file:\n");
    fp2 = fopen(filename, "wb");                                /*以只写方式打开指定文件*/
for (i = 0; i < n; i++)
     /*将数组中的员工工资信息输出到磁盘文件上*/
     fwrite(&emp[i], sizeof(struct emploee), 1, fp2);
    fclose(fp2);
    fp2 = fopen(filename, "rb");                                /*以只读方式打开指定二进制文件*/
    for (i = 0; fread(&emp[i], sizeof(struct emploee), 1, fp2) != 0; i++)/*以只读方式打开指定二进制文件
*/
        printf("\n%8s%7d", emp[i].name, emp[i].salary);         /*输出员工工资信息*/
    fclose(fp2);
}
```

程序运行结果如图 11.24 所示。

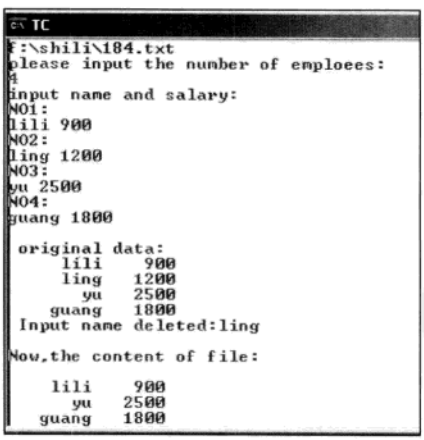


图 11.24　删除文件中的记录

技巧 16　查找文件

使用 searchpath 函数实现文件的查找，首先输入一个文件名，若该文件存在，则输出其路径，否则提示没有找到。

searchpath 函数实现找出由 fname 所指向的文件名。该函数原型在“dir.h”头文件中。

```
char *searchpath(char *fname)
```

- fname：设定文件名。
- 返回值：如果找到文件，返回指向全路径名的指针。如果文件没有找到，返回一个空指针。

例 11.16 查找文件（光盘位置：光盘\MR\Instance\11\16\16.c），该实例的主要代码如下：

```
#include<stdio.h>
main()
{
    char *ptr, filename[50];                                    /*定义变量及数组为字符型*/
    printf("please input the file name you want to search:\n");
    scanf("%s", filename);                                      /*输入要查找的文件名*/
    if ((ptr = searchpath(filename)) != NULL)
        printf("the path is:%s\n", ptr);                        /*将文件所在的路径输出*/
    els
        printf("cannot find");                                  /*若文件未找到，输出双引号内提示信息*/
}
```

程序运行结果如图 11.25 所示。

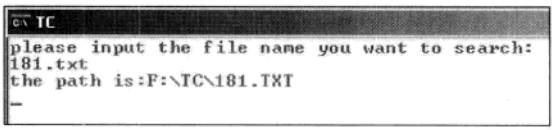


图 11.25 查找文件

技巧 17 显示目录内同类型文件

要显示目录内同类型文件需要调用多个函数来实现。首先使用 chdir()函数变换当前路径，然后调用 findfirst()查找与输入文件类型相匹配的第一个文件名，用 i 来表示 findfirst()的返回值，当 i=0 时，说明查找到第一个文件，否则说明该目录下没有与输入的文件类型相匹配的文件，输出提示信息后便退出程序。若存在第一个文件再使用 findnext()函数继续进行搜索，程序中用 j 来表示 findnext()的返回值，当 j 恒等于 0 时则继续进行下次查找，当 j 不等于 0 时，说明查找结束。

（1）chdir 函数。

该函数实现把路径名由 path 所指的目录变成当前工作目录，该函数的原型包含在“dir.h”头文件中。

```
int chdir(char *path)
```

- path：文件路径。
- 返回值：如果成功，返回 0，否则返回-1。

说明：值得注意就是 chdir 函数所设置的这个目录必须是已存在的。

（2）findfirst 函数。

该函数是寻找与 filename 所指向的文件名相匹配的第一个文件名。语法如下：

```
int findfirst(char *filename,struct ffblk *ptr,int attrib)
```

- filename：指定要查找的文件名。
- ptr：指向 ffblk 结构的指针。
- attrib：决定 findfirst 函数要找的文件的类型，FA_RDONLY 表示读文件，FA_HIDDEN 表示隐含文件，FA_SYSTEM 表示系统文件。

（3）findnext 函数。

该函数是移动文件列表指针到下一项，其中文件列表是由 findfirst 函数查找出文件列表。语法如下：

```
int findnext(struct ffblk *ptr)
```

- ptr：指向 ffblk 结构的指针。

例 11.17　显示目录内同类型文件（光盘位置：光盘\MR\Instance\11\17\17.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <dir.h>
#include <dos.h>
main()
{
    char str1[50], str2[50], filename[50];          /*定义数组为字符型*/
    int i, j = 0, k;                                /*定义变量为基本整型*/
    FILE *fp;                                       /*定义一个指向 FILE 类型结构体的指针变量*/
    struct ffblk fileinf;
    printf("please input filename which do you want to save:");
    gets(filename);
    /*输入磁盘文件路径及名称，该文件用来保存统计出的结果*/
    fp = fopen(filename, "w+");
    printf("Please input directory:\n");            /*提示输入目录*/
    gets(str1);                                     /*输入要进行统计的文件所在目录*/
    k = chdir(str1);                                /*切换当前工作目录,正确返回 0*/
    if (k != 0)                                     /*如果不等于 0,提示信息后退出*/
    {
        printf("Change Path failure!\n");
        getch();
        exit( - 1);
    }

    printf("\nPlease input File type:\n");
    gets(str2);                                     /*输入要进行统计的文件的类型，如：*.doc*/
    i = findfirst(str2, &fileinf, FA_ARCH);         /*查找匹配条件的第一个文件,正确返回 0*/
    if (i != 0)                                     /*如果返回值不为 0,提示信息后退出*/
```

```
    {
        printf("File Not Find!\n");
        getch();
        exit( - 1);
    }
    else
    {
        printf("\n");
        printf("%s\n", fileinf.ff_name);                    /*在屏幕上打印*/
        fputs(fileinf.ff_name, fp);                         /*输出到文件*/
        fputs("\n", fp);                                    /*输出换行符到文件*/
        while (j == 0)                                      /*循环查找,直到 findnext 不等于 0*/
        {
            j = findnext(&fileinf);                         /*查找与 findfirst 相匹配的文件*/
            if (j == 0)
             /*如果找到*/
            {
                printf("%s\n", fileinf.ff_name);            /*在屏幕上打印*/
                fputs(fileinf.ff_name, fp);                 /*输出到文件*/
                fputs("\n", fp);                            /*输出换行符到文件*/
            }
        }
    }
    fclose(fp);                                             /*关闭文件*/
}
```

程序运行结果如图 11.26 所示。

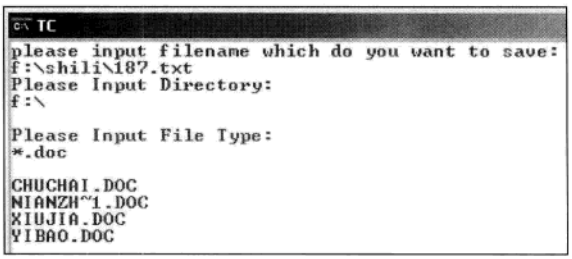


图 11.26　显示目录内同类型文件

技巧 18　关闭打开的所有文件

关闭打开的所有文件可以通过 fcloseall 函数来实现，在程序中打开 3 个磁盘上已有的文件，读取文件中的内容并显示在屏幕上，然后调用 closefileall()函数一次关闭打开的 3 个文件。

提示：打开可以使用 while 循环来依次打开，然后分别读取每个文件中的内容。

fcloseall 函数实现一次关闭所有被打开的文件。该函数原型在 stdio.h 中。

```
int fcloseall ( void )
```

- 返回值：如果函数执行成功，它将返回成功关闭文件的数目，如果出错，则返回 EOF 常量。

例 11.18　关闭打开的所有文件（光盘位置：光盘\MR\Instance\11\18\18.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    FILE *fp1,  *fp2,  *fp3;                          /*定义文件类型指针 fp1、fp2、fp3*/
    char file1[20], file2[20], file3[20], ch;
    int file_number;                                  /*关闭的文件数目*/
    printf("please input file1:");
    scanf("%s", file1);                               /*输入文件 1 的路径及名称*/
    printf("file1:\n");
    if ((fp1 = fopen(file1, "rb")) != NULL)
    {
        ch = fgetc(fp1);                              /*读取文件 1 中内容*/
        while (ch != EOF)
        {
            putchar(ch);
            ch = fgetc(fp1);
        }
    }
    else
    {
        printf("can not open!");                      /*若文件未打开输出提示信息*/
        exit(1);
    }
    printf("\nplease input file2:");
    scanf("%s", file2);                               /*输入文件 2 的路径及名称*/
    printf("file2:\n");
    if ((fp2 = fopen(file2, "rb")) != NULL)
    {
        ch = fgetc(fp2);                              /*读取文件 2 中内容*/
        while (ch != EOF)
        {
            putchar(ch);
            ch = fgetc(fp2);
        }
    }
    else
    {
        printf("can not open!");
        exit(1);
    }
    printf("\nplease input file3:");
    scanf("%s", file3);                               /*输入文件 3 的路径及名称*/
    printf("file3:\n");
    if ((fp3 = fopen(file3, "rb")) != NULL)
    {
        ch = fgetc(fp3);                              /*读取文件 3 中内容*/
        while (ch != EOF)
        {
            putchar(ch);
            ch = fgetc(fp3);
        }
    }
    else
    {
        printf("can not open!");
```

```
        exit(1);
    }
    file_number = fcloseall(); /*调用 fcloseall()函数关闭打开的文件，将返回值赋给 file_number*/
    printf("\n%d files colsed", file_number);
}
```

程序运行结果如图 11.27 所示。

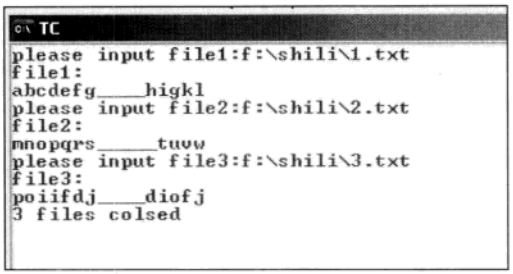


图 11.27　关闭打开的所有文件

技巧 19　错误处理

C 语言提供了错误检测函数，ferror()函数可以进行错误检查。将文件中的制表符换成恰当数目的空格时，可以调用 ferror()函数检查错误。

ferror 函数是检测已打开文件流里的文件错误。

```
int ferror(FILE *stream)
```

- stream：FILE 类型文件指针。
- 返回值：返回值为 0 时，表示没有出现错误；而非零值表示有错。

提示：ferror 函数的原型在 stdio.h 中，与 stream 相关联的出错标记给出后，一直要保持到该文件被关闭，或调用了 rewind 或者 clearerr 为止。使用 perror 函数可以确定该错误的确切性质。

例 11.19　错误处理（光盘位置：光盘\MR\Instance\11\19\19.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <stdlib.h>
//自定义 error()函数，作用是输出出错的性质
void error(int e)                                    /*自定义 error 函数判断出错的性质*/
{
    if(e == 0)
        printf("input error\n");
    else
        printf("output error\n");
    exit(1);                                         /* 跳出程序 */
}
//main()函数作为程序的入口函数
main()
```

```
{
    FILE *in,  *out;                                    /*第一两个文件类型指针 in 和 out*/
    int tab, i;
    char ch, filename1[30], filename2[30];
    printf("please input the filename1:");
    scanf("%s", filename1);                             /*输入文件路径及名称*/
    printf("please input the filename2:");
    scanf("%s", filename2);                             /*输入文件路径及名称*/
    if ((in = fopen(filename1, "rb")) == NULL)
    {
        printf("can not open the file %s。\n", filename1);
        exit(1);
    }
    if ((out = fopen(filename2, "wb")) == NULL)
    {
        printf("can not open the file %s。\n", filename2);
        exit(1);
    }
    tab = 0;
    ch = fgetc(in);                                     /*从指定的文件中读取字符*/
    while (!feof(in))
                                                        /*检测是否有读入错误*/
    {
        if (ferror(in))
            error(0);
        if (ch == '\t')
                                                        /*如果发现制表符，则输出相同数目的空格符*/
        {
            for (i = tab; i < 8; i++)
            {
                putc(' ', out);
                if (ferror(out))
                    error(1);
            }
            tab = 0;
        }
        else
        {
            putc(ch, out);
            if (ferror(out))
                                                        /*检查是否有输出错误*/
                error(1);
            tab++;
            if (tab == 8)
                tab = 0;
            if (ch == '\n' || ch == '\r')
                tab = 0;
        }
        ch = fgetc(in);
    }
    fclose(in);
    fclose(out);
}
```

程序运行结果如图 11.28 所示。

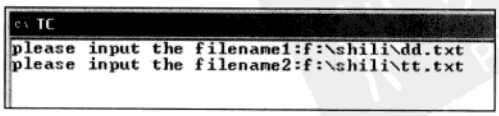


图 11.28　错误处理

程序运行时需要用户输入两个文件，一个文件名为“dd.txt”，文件内容如图 11.29 所示。一个文件名

为“TT.TXT”，文件内容如图 11.30 所示。

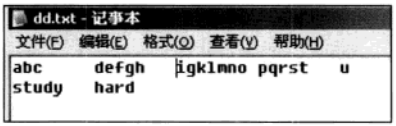


图 11.29 dd.txt 内容

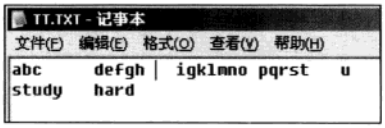


图 11.30 TT.TXT 内容



第 12 章　数学应用开发技巧

学习计算机或数学专业的人都知道这样一句话：“计算机和数学不分家”，这句话很好地说明了计算机和数学的关系是十分紧密的，数学中的很多问题都可以通过编程来解决。

在学习过程中会发现许多数之间都有着内在联系，如果想找出存在相同规律的一组数光凭用笔来算会花费大量的时间，且很容易漏掉某个数，如果用编程的方法来找具有相同规律的一组数，不但简单、方便、快捷而且不会漏掉任何一个数。

通过本章的学习，读者可以了解有规律数的求法、掌握常用数学公式的计算方法、矩阵运算以及解决日常生活中的问题。

在本章我们将讲述一些最基本的 SQL 内容，如 SQL 的历史起源、ANSI 是什么，以及 SQL 的语法元素和执行 SQL 的工具等。对于基本知识，学习起来可能比较枯燥。但是，如果你是初学 SQL，这些基本知识对于学习好本书的后续内容，将起着至关重要的作用。

技巧 01　求 100～200 之间的素数

素数是大于 1 的整数，除了能被自身和 1 整除外，不能被其他正整数整除。100～200 之间的全部素数可以通过如下算法实现：假如有整数 i，让 i 被 2 到根号 i 除，如果 i 能被 2 到根号 i 之中任何一个整数整除，则结束循环，若不能被整除则要判断 j 是否是最接近或等于根号 i 的，如果是则证明是素数，否则继续下次循环。

sqrt 函数的作用是返回 x 的开方值。该函数原型定义在头文件 math.h 中。语法如下：

```
double sqrt(double x)
```

- x：设置需要开方的数值。
- 返回值：返回计算结果。

例 12.1　求 100～200 之间的素数（光盘位置：光盘\MR\Instance\12\01\ 01.c），该实例的主要代码如下：

```
main()
{
    int i, j, n = 0;                                    /*定义变量为基本整型*/
    clrscr();
```

```
    for (i = 100; i <= 200; i++)
        for (j = 2; j <= sqrt(i); j++)
            if (i % j == 0)                          /*判断是否能被整除*/
                break;                               /*如果能被整除，就不需要接着判断，跳出循环*/
            else
    if (j > sqrt(i) - 1)
    {
        printf("%d,", i);
        n++;                                         /*记录次数*/
        if (n % 5 == 0)                              /*5 个一换行*/
            printf("\n");
    }
    else
        continue;
}
```

```
TC
101,103,107,109,113,
127,131,137,139,149,
151,157,163,167,173,
179,181,191,193,197,
199,_
```

图 12.1　求 100~200 之间的素数

程序运行结果如图 12.1 所示。

技巧 02　可逆素数

可逆素数是指将一个素数的各位数字顺序倒过来构成的反序数仍然是素数。计算四位素数可以使用逐位提取数字的方法，即对这个四位数采用“%”或“/”的方法分解出每位上的数字。再对重新组合的数字进行判断看是否是素数并判断是否大于原来的四位数，这样做主要是为了防止重复输出，例如 1031 与 1301 互可逆素数，这里我们只输出 1031 即可，1301 不必输出。

例 12.2　可逆素数（光盘位置：光盘\MR\Instance\12\02\02.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <math.h>
//自定义 ss 函数，作用是判断一个数是否为素数
int ss(int i)                                        /*自定义函数判断是否为素数*/
{
    int j;
    if (i <= 1)                                      /*小于 1 的数不是素数*/
        return 0;
    if (i == 2)                                      /*2 是素数*/
        return 1;
    for (j = 2; j < i; j++)                          /*对大于 2 的数进行判断*/
    {
        if (i % j == 0)
            return 0;
        else if (i != j + 1)
            continue;
        else
            return 1;
    }
}
//对 1000~10000 的数进行穷举，找出符合条件的数并将其输出
main()
{
    int i, n = 0, n1, n2, n3, n4;
    clrscr();
    for (i = 1000; i < 10000; i++)
    if (ss(i) == 1)
    {
        n4 = i % 10;                                 /*取个位数*/
        n3 = (i % 100) / 10;                         /*取十位数*/
```

```
        n2 = (i / 100) % 10;                                          /*取千位数*/
        n1 = i / 1000;                                                /*取万位数*/
        if (ss(1000 *n4 + 100 * n3 + 10 * n2 + n1) == 1 && 1000 *n4 + 100 * n3
            + 10 * n2 + n1 > i)                                       /*根据条件判断*/
        {
            printf("%d,", i);
            n++;                                                      /*记录个数*/
            if (n % 10 == 0)                                          /*10 个一换行*/
                printf("\n");
        }
    }
}
```

程序运行结果如图 12.2 所示。

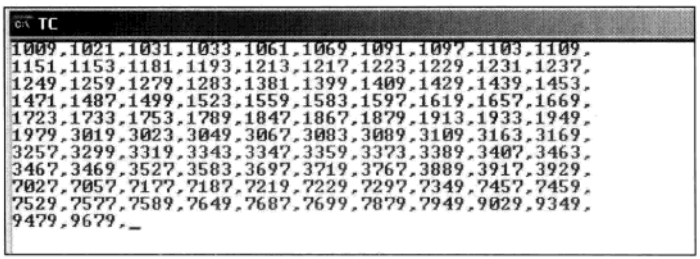


图 12.2　可逆素数

技巧 03　回文素数

任意整数 i，当从左向右的读法与从右向左的读法是相同且为素数时则称该数为回文素数，求 1000 之内的所有回文素数的步骤如下：首先要判断该素数是两位数还是三位数，若是两位数只需判断每位上的数字是否相等便可，若是三位数则需判断个位上和百位上的数字是否相同，相同表明该素数是回文素数，不同则继续下次判断。

例 12.3　回文素数（光盘位置：光盘\MR\Instance\12\03\03.c），该实例的主要代码如下：

```
#include <stdio.h>
//自定义 ss 函数，判断一个数是否为素数
int ss(int i)
{
    int j;
    if (i <= 1)                                                       /*小于 1 的数不是素数*/
        return 0;
    if (i == 2)                                                       /*2 是素数*/
        return 1;
    for (j = 2; j < i; j++)                                           /*对大于 2 的数进行判断*/
    {
        if (i % j == 0)
            return 0;
        else if (i != j + 1)
            continue;
        else
            return 1;
    }
}
```

```
//对 10~1000 的数进行穷举，找出符合条件的数并将其输出
main()
{
    int i;
    for (i = 10; i < 1000; i++)
        if (ss(i) == 1)                                    /*判断是否是素数*/
    if (i / 100 == 0)                                      /*判断是否是两位数*/
    {
        if (i / 10 == i % 10)                              /*判断十位和各位是否相同*/
                printf("%5d", i);
    }
    else
        if (i / 100 == i % 10)                             /*判断百位和个位是否相同*/
            printf("%5d", i);
}
```

程序运行结果如图 12.3 所示。

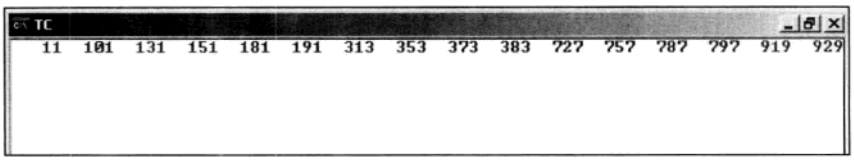


图 12.3 回文素数

技巧 04 特殊的完全平方数

完全平方数主要指一个整数乘以自己，特殊完全平方数主要指既是完全平方数，同时还满足整个数中有两位数字相同的数，如 121、144 等。求特殊的完全平方数的主要思想如下：对探测到的 100～999 之间的数首先要判断它是不是完全平方数，如果是完全平方数再分离出其百位、十位、个位上的数字，用 if 条件判断语句判断分离出的 3 个数中是否有两个数相同。如果有两个数相同则输出该数字，否则继续下次循环。如果不是完全平方数就不用分离再判断直接进行下次探测便可。

提示：可以用 while 循环判断探测到的数据是否是完全平方数。

例 12.4 特殊的完全平方数（光盘位置：光盘\MR\Instance\12\04\04.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int i, j;
    int hun, ten, data; /*定义变量存储分解出的百位、十位、个位*/
    printf("the result is:\n");
    for (i = 100; i <= 999; i++)
    {
        j = 10;
        while (j *j <= i)
        {
              if (i == j *j)
```

```
            {
                hun = i / 100;                                  /*分解出百位上的数*/
                data = i - hun * 100;
                ten = data / 10;                                /*分解出十位上的数*/
                data = data - ten * 10;                         /*分解出个位上的数*/
                /*判断分解出的 3 个数中是否有 2 个数是相等的*/
                if (hun == ten || hun == data || ten == data)
                printf("%5d", i);                               /*将符合条件的数输出*/
            }
            j++;
        }
    }
}
```

程序运行结果如图 12.4 所示。

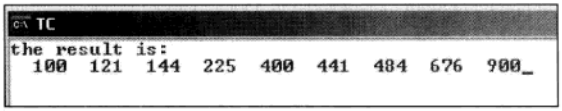


图 12.4　特殊的完全平方数

技巧 05　求 1000 以内的完数

一个数如果恰好等于它的因子之和，这个数就称为“完数”。例如 6 的因子是 1、2、3，而 6=1+2+3，因此 6 是“完数”。可以采用穷举的方法求 1000 以内的完数，首先对 1000 以内的数逐个求因子再求和看是否等于该数，如果相等则输出该数，否则进行下次循环。这里要强调的是求一个数 *i* 因子的方法即用该数 *i* 对从 1 到 *i*-1 的所有数取余，看余数是否为 0，如果为 0 则说明该数是 *i* 的因子。

例 12.5　求 1000 以内的完全数（光盘位置：光盘\MR\Instance\12\05\05.c），该实例的主要代码如下：

```
#include <stdio.h>
//对 1 到 1000 之间的数按照完数的要求进行运算并判断，并将最终符合要求的结果输出。
main()
{
    int i, j, sum = 0;                                  /*定义变量为基本整型*/
    for (i = 1; i < 1000; i++)                          /*对 1000 以内的数进行穷举*/
    {
        sum = 0;
        for (j = 1; j < i; j++)
            if (i % j == 0)                             /*判断 j 是否是 i 的因子*/
                sum += j;                               /*因子相加求和*/
        if (sum == i)                                   /*判断该数是否等于各因子之和*/
            printf("%4d", i);
    }
}
```

程序运行结果如图 12.5 所示。

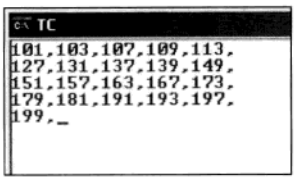


图 12.5　求 1000 以内的完数

技巧 06　三重回文数

回文数指其各位数字左右对称的整数，三重回文数应满足对于整数 n，n、n^2、n^3 均为回文数。

判断回文数可以不适用逐位分离再判断的方法，可以将一个数转换成字符串存到字符型数组 a 中，再对这个数组做判断看其是否在对应位置相等，如在对应位置均相等则说明是回文数，只要有一个不相等则说明不是回文数。要将整型数转换为字符串，可以使用函数 ltoa。

ltoa 函数：函数的作用是把长整数 num 转换成其等价的字符串，并且把结果保存到由 str 所指向的字符串中，字符串输出的进制数是由 radix 确定的，它一般在 2～36 的范围内变化。该函数的原型在 stdlib.h 中。函数原型如下：

```
char *ltoa(long num,char *str,int radix)
```

- num：指定转换的个数。
- str：设置字符指针。
- radix：设置进制属性。
- 返回值：指向 str 的指针。

说明：实例计算出 10～1000 之间的三重回文数。

例 12.6　三重回文数（光盘位置：光盘\MR\Instance\12\06\06.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <stdlib.h>
//自定义 judge 函数，用来判断一个数是否是回文数
int judge(long n)                                    /*判断一个数是否是回文数*/
{
    int i, len, half;                                /*定义变量为基本整型*/
    char a[20];                                      /*定义数组 a 为字符型*/
    ltoa(n, a, 10);                                  /*把长整形数 n 转换成字符串存到数组 a 中*/
    len = strlen(a);                                 /*获取数组 a 的长度*/
    half = len / 2;
    for (i = 0; i < half; i++)
        if (a[i] != a[--len])
                                                     /*判断相对应位置的字符是否相同*/
```

```
            break;
                                                    /*如果不相同跳出循环*/
    if (i >= half)
        return 1;
                                                    /*是回文数返回 1*/
    else
        return 0;
                                                    /*不是回文数返回 0*/
}
main()
{
    long n;
    clrscr();
    printf("the result is:\n");
    for (n = 11; n < 1000; n++)                     /*对 10~1000 之间的数进行判断*/
        if (judge(n) && judge(n *n) && judge(n *n * n))    /*判断是否是三重回文数*/
            /*将三重回文数输出*/
            printf(" n=%4ld,n*n=%6ld,n*n*n=%8ld \n", n, n *n, n *n * n);
    getch();
}
```

程序运行结果如图 12.6 所示。

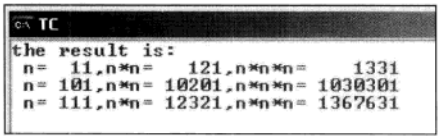


图 12.6　三重回文数

技巧 07　亲密数

如果整数 A 的全部因子（不包括 1 和 A）之和等于 B，且整数 B 的全部因子（不包括 1 和 B）之和等于 A，则将 A 和 B 称为亲密数。可以采用穷举的方法求出 10000 以内的亲密数，首先对 10000 以内的数逐个求因子，并求出所有因子之和 sum1，再对所求出的和 sum1 求因子，并再次求所有因子之和 sum2，此时按亲密数的要求进行进一步筛选便求出最终结果。

提示：这里求因子的方法依旧是用数 i 对从 1 到 i-1 的所有数取余，看余数是否为 0，如果为 0，则说明该数是 i 的因子。

例 12.7　亲密数（光盘位置：光盘\MR\Instance\12\07\07.c），该实例的主要代码如下：

```
#include <stdio.h>
//对 1 到 10000 之间的数进行判断，并将最终符合要求的结果输出
main()
{
    int i, j, k, sum1, sum2;                        /*定义变量为基本整型*/
    for (i = 1; i <= 10000; i++)                    /*对 10000 以内的数进行穷举*/
    {
        sum1 = 0;
        sum2 = 0;
```

```
        for (j = 1; j < i; j++)
            if (i % j == 0)                          /*判断 j 是否是 i 的因子*/
                sum1 += j;                           /*求因子的和*/
        for (k = 1; k < sum1; k++)
            if (sum1 % k == 0)                       /*判断 k 是否是 sum1 的因子*/
                sum2 += k;                           /*求因子和*/
        if (sum2 == i && i != sum1 && i < sum1)
            printf("\n%5d=>%5d", i, sum1);           /*将亲密数输出*/
    }
}
```

程序运行结果如图 12.7 所示。

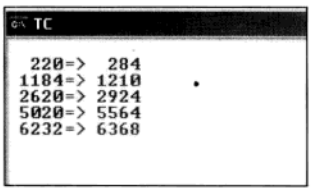


图 12.7 亲密数

技巧 08 自守数

自守数是指一个数平方的尾数等于该数自身的自然数。下列数字便是自守数：25^2=625，76^2=5776，9376^2=87909376。可以通过手工求解的方法，来分析求解 100000 以内的所有自守数的方法，计算公式如下：

$$\begin{array}{r} 9376 \\ \times\ 9376 \\ \hline 56256 \\ 65632 \\ 28128 \\ 84384 \\ \hline 87909376 \end{array}$$

观察上式发现 9376×9376 的最终结果是 87909376，这里我们关心积的后 4 位是如何产生的，具体规律如下：

- 第 1 部分积（56256）：被乘数的最后 4 位乘以乘数的倒数第 1 位。
- 第 2 部分积（65632）：被乘数的最后 3 位乘以乘数的倒数第 2 位。
- 第 3 部分积（28128）：被乘数的最后 2 位乘以乘数的倒数第 3 位。
- 第 4 部分积（84384）：被乘数的最后 1 位乘以乘数的倒数第 4 位。

将以上 4 部分的后 4 位求和后取后 4 位便可求出一个 4 位数乘积的后 4 位。其他不同位数的数依此类推。

例 12.8　自守数（光盘位置：光盘\MR\Instance\12\08\08.c），该实例的主要代码如下：

```
#include<stdio.h>
//对 1 到 10000 之间的数按照自守数规则进行判断，并将最终符合要求的结果输出
main()
{
    long i, j, k1, k2, k3, a[10] =
    {
        0
    }
    , num, m, n, sum;                                    /*定义变量及数组为长整形*/
    printf("please inpput a number:\n");
    scanf("%ld", &num);                                  /*从键盘中输入要求的范围*/
    printf("the result is:");
    for (j = 0; j < num; j++)                            /*对该范围内的数逐个试探*/
    {
        m = j;
        n = 1;
        sum = 0;
        k1 = 10;
        k2 = 1;
        while (m != 0)                                   /*判断该数的位数*/
        {
            a[n] = j % k1;                               /*分离出的数存入数组中*/
            n++;                                         /*记录位数，实际位数为 n-1*/
            k1 *= 10;                                    /*最小 n 位数*/
            m = m / 10;
        }
        k1 = k1 / 10;
        k3 = k1;
        for (i = 1; i <= n - 1; i++)
        {
            sum += (a[i] / k2 * a[n - i]) % k1 * k2;     /*求每一部分积之和*/
            k2 *= 10;
            k1 /= 10;
        }
        sum = sum % k3;                                  /*求和的后 n-1 位*/
        if (sum == j)
            printf("%8ld", sum);                         /*输出找到的自守数*/
    }
}
```

程序运行结果如图 12.8 所示。

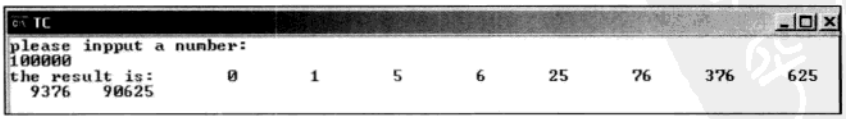


图 12.8　自守数

技巧 09　满足 abcd=（ab+cd）^{2} 的数

假设 abcd 是一个 4 位整数，将它分成两段，即 ab 和 cd，使之相加求和后再平方。可以采

用穷举的方法求得 1000 到 10000 以内满足条件 abcd=（ab+cd）^{2}的数，首先对所有 4 位整数逐个分解成两部分，然后对其进行判断，看是否满足要求，如果满足则将该整数输出，否则进行下次循环。

提示：将一个 4 位数分解成两部分主要采用“/”和“%”的方法，“/”求的该 4 位数的前两位，“%”求的是该 4 位数的后两位。

例 12.9 满足 abcd=（ab+cd）^{2}的数（光盘位置：光盘\MR\Instance\12\09\09.c），该实例的主要代码如下：

```
#include <stdio.h>
//使用 for 循环对 1000 到 10000 之间所有的 4 位正整数按照亲技术要点中所提到的过程进行分解并判断
//并将最终符合要求的结果输出。
main()
{
    int i, a, b;                                    /*定义变量为基本整型*/
    for (i = 1000; i < 10000; i++)                  /*对 1000~10000 间数进行穷举*/
    {
        a = i / 100;                                /*求出该数的前两位数*/
        b = i % 100;                                /*求出该数的后两位数*/
        if ((a + b)*(a + b) == i)                   /*判断是否满足条件*/
            printf("\n%5d", i);
    }
}
```

程序运行结果如图 12.9 所示。

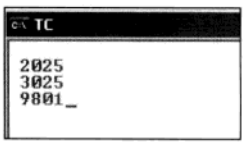


图 12.9 满足 abcd=（ab+cd）^{2}的数

技巧 10 神奇的数字 6174

神奇的数字 6174 主要指：从键盘中任意输入一个 4 位数，要求该 4 位数各个位上的数字是不全相同的，将组成该 4 位数的 4 个数字由大到小排列，形成由这 4 个数组成的最大 4 位数；再由小到大排列，形成有这 4 个数组成的最小 4 位数（如果数字中有 0，则得到不足 4 位的数）；求这个最大数与最小数的差，重复以上过程，最终得到的结果总是 6174。

神奇的数字 6174 的实现过程主要是先提取出一个 4 位数的各位，然后经过排序，放入到数组中，再根据排序后的结果生成，由大到小排列和由小到到排列的两个 4 位数，最后计算出两个数的和再进行判断，如果符合要求就输出计算过程。

例 12.10 神奇的数字 6174（光盘位置：光盘\MR\Instance\12\10\10.c），该实例的主要代码如下：

```
#include <stdio.h>
//自定义 difference 函数，实现求最大 4 位数与最小数的差
int difference(int a[])
{
    int t, i, j, sum, sum1, sum2;
    for (i = 0; i < 3; i++)                          /*利用选择排序法对数组 a 中的数据由大到小排序*/
        for (j = i + 1; j < 4; j++)
    if (a[i] < a[j])
    {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    sum1 = a[0] *1000+a[1] *100+a[2] *10+a[3];       /*将数组中的 4 个数组成最大 4 位数*/
    sum2 = a[3] *1000+a[2] *100+a[1] *10+a[0];       /*将数组中的 4 个数组成最小 n 位数，n 小于等于 4*/
    sum = sum1 - sum2;                               /*求最大数与最小数的差*/
    printf("%5d=%5d-%5d\n", sum, sum1, sum2);        /*讲求的的结果按指定格式输出*/
    return sum;                                      /*将差值返回*/
}
main()
{
    int i, j, k, l, n, a[4];
    printf("please input a number:\n");
    scanf("%d", &n);                                 /*从键盘中输入一个 4 位数*/
    while (n != 6174)
    {
        a[0] = n / 1000;                             /*分离出千位上的数*/
        a[1] = n / 100 % 10;                         /*分离出百位上的数*/
        a[2] = n / 10 % 10;                          /*分离出十位上的数*/
        a[3] = n % 10;                               /*分离出个位上的数*/
        n = difference(a);                           /*调用 difference 函数，将返回值赋给 n*/
    }
}
```

程序运行结果如图 12.10 所示。

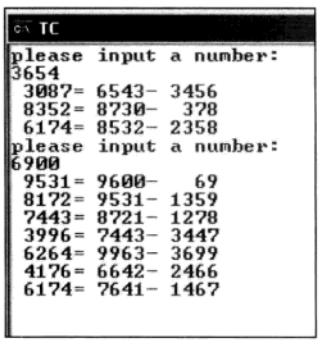


图 12.10　神奇的数字 6174

技巧 11　一数三平方

有这样一个 6 位数，它本身是一个整数的平方，其高 3 位和低 3 位也分别是一个整数的平方，例如 $225625=475^2$，$225=15^2$，$625=25^2$。求 100000 到 999999 之间的所有三平方的数据进行求解，

其思路是：利用 for 循环对所有 6 位数进行穷举，然后使用 sqrt 函数求得平方根，如果平方根是整数，就取得该数的高 3 位和低 3 位，继续使用 sqrt 函数分别求平方根，如果求得的两个平方根仍然是整数，就输出该值。

sqrt 函数

该函数是返回参数 num 的平方根，函数原型如下：

```
double sqrt(double num)
```

- num：求平方根的数。
- 返回值：返回计算结果。

提示：这里我们可以发现 sqrt 的返回值是一个 double 型，程序中我们将 sqrt 的返回值强制转换成长整型，这样会使开平方后得到的小数（小数点后不为 0）失去其小数点后面的部分，那么再对这个强制转换后的数再平方所得的结果将不会等于原来开平方前的数。若开平方后得到的小数其小数点后的部分为 0，则将其强制转换为长整型也不会产生数据流失，那么再对这个强制转换后的数再平方所得的结果就将等于原来开平方前的数。我们利用这个方法就能很好地判断出一个数开平方后得到的数是否是整数。

例 12.11 一数三平方（光盘位置：光盘\MR\Instance\12\11\11.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <math.h>
main()
{
    long i, n, n1, n2, n3, n4, count = 0;                    /*定义变量为长整型*/
    printf("the result is:\n");
    for (i = 100000; i <= 999999; i++)                       /*遍历所有的 6 位数*/
    {
        n = (long)sqrt(i);                                   /*对 i 值开平方得到一个长整型数值 n*/
        if (i == n *n)                                       /*判断 n 的平方是否等于 i*/
        {
            n1 = i / 1000;                                   /*求出高 3 位数*/
            n2 = i % 1000;                                   /*求出低 3 位数*/
            n3 = (long)sqrt(n1);                             /*对 n1 值开平方得到一个长整型数值 n3*/
            n4 = (long)sqrt(n2);                             /*对 n2 值开平方得到一个长整型数值 n4*/
            if (n1 == n3 *n3 && n2 == n4 *n4)
                    /*判断是否同时满足 n1 等于 n3 的平方，n2 等于 n4 的平方*/
            {
                count++;                                     /*count 作为计数器，记录满足条件的个数*/
                printf("%ld,", i);
            }                                                /*将最终满足条件的 i 值输出*/
        }
    }
    printf("\ntotal is :\n%d", count);                       /*输出满足条件的个数*/
}
```

程序运行结果如图 12.11 所示。

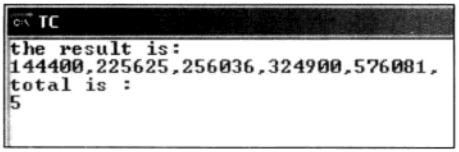


图 12.11　一数三平方

技巧 12　求等差数列

幼儿园老师给学生由前向后发糖果，每个学生得到的糖果数目成等差数列，前 4 个学生的得到的糖果数目之和是 26，积是 880，求前 20 名学生每人得到的糖果数目。这样题目的实现过程如下：要先确定这个等差数列的首项及公差的取值范围，因为数字比较小范围确定读者可通过手工来算，很容易确定出等差数列首项的范围是 1～6 之间（不包括 6），公差的取值范围是 1～4 之间（不包括 4），在确定了首项和公差之后，便可按照题意设置条件进行判断，可以使用 for 语句对数列首项和公差进行穷举，对满足 if 语句中条件的按指定的格式输出，否则进行下次循环。

例 12.12　求等差数列（光盘位置：光盘\MR\Instance\12\12\12.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int i, j, number, n;
    for (number = 1; number < 6; number++)                    /*对 1 到 5 之间的数进行穷举*/
        for (n = 1; n < 4; n++)                               /*对 1 到 3 之间的数进行穷举*/
    if ((4 *number + 6 * n == 26) && (number *(number + n)*(number + 2 * n)*
        (number + 3 * n)) == 880)                             /*判断是否满足题中条件*/
    {
        printf("The result is:\n");
        for (j = 1; j <= 20; j++)
        {
            printf("%3d", number);
            number += n;
            if (j % 5 == 0)                                   /*每输出 5 个进行换行*/
                    printf("\n");
        }
    }
}
```

程序运行结果如图 12.12 所示。

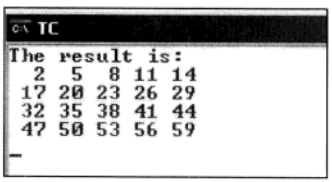


图 12.12　求等差数列

技巧 13 求整数的绝对值

C 语言中提供了 abs、labs 和 fabs 等函数用来求得符号整数的绝对值，可以实现从键盘上任意输入一个整数，然后在屏幕中显示计算结果。

abs 函数

该函数的功能是求整数的绝对值。函数原型：

```
int abs(int x)
```

- x：要计算绝对值的整型数。
- 返回值：返回整型的计算结果。

labs 函数

该函数的功能是求长整型整数的绝对值。函数原型：

```
long labs (long x)
```

- x：要计算绝对值的长整型数。

fabs 函数

该函数的功能是求浮点数的绝对值，函数原型：

```
double fabs (double x)
```

- x：要计算绝对值的双精度数。

例 12.13 巧分苹果（光盘位置：光盘\MR\Instance\12\13\13.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <math.h>
main()
{
    int i;                                              /*定义变量 i 为基本整型*/
    printf("please input a number:\n");                 /*双引号内普通字符原样输出并回车*/
    scanf("%d", &i);                                    /*从键盘中输入数值并赋值给 i*/
    printf("number: %d ,absolute value: %d ", i, abs(i)); /*调用 abs 函数求出绝对值*/
}
```

程序运行结果如图 12.13 所示。

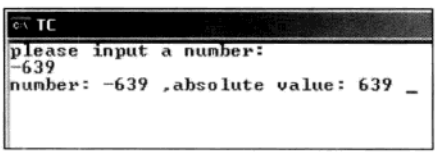


图 12.13 巧分苹果

技巧 14　正弦、余弦、正切值

编程实现从键盘上任意输入一个整数，在屏幕中显示出其相应的正弦值、余弦值以及正切值。运行结果如图 5.15 所示。

sin 函数

该函数的功能是返回参数 x 的正弦值。函数原型：

```
double sin(double x)
```

- x：将要求正弦值的弧度。

cos 函数

该函数的功能是返回参数 x 的余弦值。函数原型：

```
double cos(double x)
```

- x：将要求余弦值的弧度。

tan 函数

该函数的功能是返回参数 x 的正切值。函数原型：

```
double tan (double x)
```

- x：将要求正切值的弧度。

例 12.14　巧分苹果（光盘位置：光盘\MR\Instance\12\14\14.c），该实例的主要代码如下：

```
#include <stdio.h>
#include <math.h>
main()
{
    float n;                                          /*定义 n 为单精度型*/
    printf("please input:\n");                        /*双引号内普通字符原样输出并回车*/
    scanf("%f", &n);                                  /*从键盘中输入一个单精度数并赋给 n*/
    printf("the sin of %f is %f\n", n, sin(n));       /*调用 sin 函数输出正弦值*/
    printf("the cos of %f is %f\n", n, cos(n));       /*调用 cos 函数输出余弦值*/
    printf("the tan of %f is %f\n", n, tan(n));       /*调用 tan 函数输出正弦值*/
}
```

程序运行结果如图 12.14 所示。

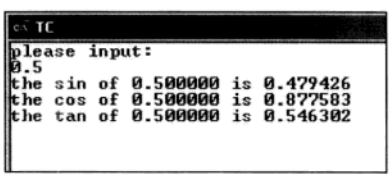


图 12.14　正弦、余弦、正切值

技巧 15 自然对数的底 e 的计算

自然对数的底 e=2.718281828…，e 的计算公式是：

$$e=1+1/1!+1/2!+1/3!+\cdots$$

仔细观察 e 的计算公式会发现，想求出最终结果的关键是求每一项所对应的阶层。可以使用 while 循环，循环体中实现累加求和，并在求和之后求下一项的所对应的阶层。

提示：当最后一项的值小于 1e-10 时结束循环。

例 12.15 自然对数的底 e 的计算（光盘位置：光盘\MR\Instance\12\15\ 15.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    float e = 1.0, n = 1.0;                 /*定义 e 和 n 为单精度型，并为它们赋初值为-1*/
    int i = 1;                              /*定义 i 为基本整型并赋初值为 1*/
    while (1 / n > 1e-10)                   /*当该项的值不小于 1e-10 时，执行循环体中内容*/
    {
        e += 1 / n;                         /*累加各项的和*/
        i++;
        n = i * n;                          /*求阶层*/
    }
    printf("the value of e is %f", e);      /*将最终结果输出*/
}
```

```
TC
the value of e is 2.718282
```

程序运行结果如图 12.15 所示。

图 12.15 自然对数的底 e 的计算

技巧 16 最大公约数及最小公倍数

最大公约数及最小公倍数是数学中经常用到的，最大公约数及最小公倍数需要根据两个数进行计算，这两个数可以通过用户输入获得，用户从键盘中输入两个正整数 a 和 b，可以求出 a 和 b 的最大公约数和最小公倍数。

在算两个数的最大公约数时，我们通常采用辗转相除的方法，使用 scanf 函数将输入的两个数分别赋值给 a 和 b，如果 a 小于 b，则借助中间变量 t 将 a 与 b 的值互换，用辗转相除的方法求出 a 和 b 的最大公约数。

最小公倍数和最大公约数之间的关系是：两数相乘的积除以这两个数的最大公约数就是最小公倍数。知道这层关系后用辗转相除法求出最大公约数，那么最小公倍数也便求出来了。

例 12.16 最大公约及最小公倍数（光盘位置：光盘\MR\Instance\12\16\16.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int a, b, c, m, t;                                  /*定义变量为基本整型*/
    printf("please input two numbers:\n");
    scanf("%d%d", &a, &b);                              /*从键盘中输入两个数*/
    if (a < b)                                          /*当 a 小于 b 时实现俩值互换*/
    {
        t = a;
        a = b;
        b = t;
    }
    m = a * b;                                          /*求出 a 与 b 的乘积*/
    c = a % b;                                          /*a 对 b 取余赋给 c*/
    while (c != 0)                                      /*当 c 不为 0 时执行循环体语句*/
    {
        a = b;                                          /*将 b 赋给 a*/
        b = c;                                          /*将 c 的值赋给 b*/
        c = a % b;                                      /*继续取余并赋给 c*/
    }
    printf("the max commen divisor:\n%d\n", b);         /*输出最大公约数*/
    printf("the min commen multiple:\n%d", m / b);      /*输出最小公倍数*/
}
```

程序运行结果如图 12.16 所示。

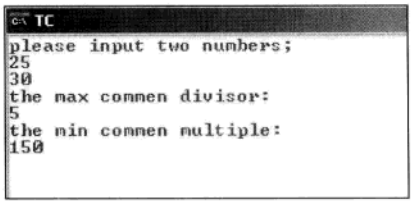


图 12.16　最大公约及最小公倍数

技巧 17　求解二元一次不定方程

二元一次不定方程是数学中基础方程。可以实现求二元一次不定方程 $ax+by=c$ 的解，其中 a、b、c 由用户从键盘中输入，而且满足 $a>0$，$b>0$ 且 $a>=b$。

关于求解二元一次不定方程 $ax+by=c$ 有两个重要的结论：

- 二元一次不定方程有解的充分必要条件是 a 与 b 的最大公约数能整除 c。
- 如果（$x0$,$y0$）是方程的一组解，则对任何整数 k，（$x0+bk$，$y0-ak$）也都是方程的解。

下面以 $123x+35y=7$ 为例来分析下二元一次不定方程的求解算法：

$$123=35*3+18$$

$$35=18*1+17$$

$$18=17*1+1$$

$$17=1*17$$

将上述推导过程用公式表示出来便是：

$$an=bn*cn+dn，\ cn=an/bn，\ dn=an\%bn，\ an+1=bn，\ bn+1=dn$$

$x[n]$,$y[n]$的递推计算公式如下：

$$x[0]=0,x[1]=1,x[n+1]=x[n]*q[n]+x[n-1] \quad (i>1)$$

$$y[0]=1,y[1]=q[0],y[n+1]=y[n]*q[n]+y[n-1] \quad (i>1)$$

最终方程的解便是 $x=(-1)n-1x[n],y=(-1)ny[n]$，这里的 n 的值就是进行计算的轮数。

例 12.17 求解二元一次不定方程（光盘位置：光盘\MR\Instance\12\17\ 17.c），该实例的主要代码如下：

```
#include <stdio.h>
//自定义 result 函数，用来求解二元一次不定方程的一组解
void result(int a, int b, int c, int *x2, int *y2)              /*自定义函数解*/
{
    int x[100], y[100], z[100];
    int i, j, d, t, gcd;
    x[0] = 0;
    y[0] = 1;
    for (i = 0; i < 100; i++)
    {
        z[i] = a / b;                                           /*求 a/b 的值*/
        d = a % b;                                              /*求 a 对 b 取余的值*/
        a = b;
        b = d;
        if (d == 0)
        {
            gcd = a;                                            /*辗转法求最大公约数*/
            break;
        }
        if (i == 0)                                             /*判断 a 是否能被 b 整除*/
        {
            x[1] = 1;
            y[1] = z[0];
        }
        else
        {
            x[i + 1] = z[i] *x[i] + x[i - 1];
            y[i + 1] = z[i] *y[i] + y[i - 1];
        }
    }
    for (t =  - 1, j = 1; j < i; j++)
        t =  - t;
    *x2 =  - t * x[i];
    *y2 = t * y[i];                                             /*求出 ax+by=gcd(a, b)的一组解*/
    if (c % gcd != 0)                                           /*如果 c 能否整除 a 和 b 的最大公约数*/
    {
        printf("No solution!\n");                               /*如不能整除，则输出无解的提示信息*/
        exit(0);
    }
    t = c / gcd;                                                /*若能整除则将结果赋给 t*/
```

```
    *x2 =  *x2 * t;
    *y2 =  *y2 * t;
}
//自定义 test 函数，用来检验求出的解是否满足方程
void test(int a, int b, int c, int x, int y)                    /*自定义函数检测求出的结果*/
{
    if (a *x + b * y == c)                                      /*将 x、y 带进整式看是否等于 c*/
        printf("Result right!\n");
    else
        printf("Result error!\n");
}
main()
{
    int a, b, c, x2, y2;
    printf("input a,b,c:\n");                                   /*输入 a、b、c 的值*/
    scanf("%d%d%d", &a, &b, &c);
    result(a, b, c, &x2, &y2);                                  /*调用函数求出解*/
    test(a, b, c, x2, y2);                                      /*检验结果是否正确*/
    printf("x=%d,y=%d\n", x2, y2);                              /*将 x、y 的值输出*/
}
```

程序运行结果如图 12.17 所示。

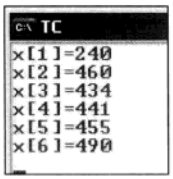


图 12.17　求解二元一次不定方程

技巧 18　二分法求解方程

二分法解方程的思路如下：如果函数 $f(x)$在指定区间$[x1,x2]$内具有单调性，当 $f(x1)$与 $f(x2)$异号时说明方程 $f(x)=0$ 在区间$[x1,x2]$内有且仅有一个实根，当 $f(x1)$和 $f(x2)$同号时方程 $f(x)=0$ 在区间$[x1,x2]$内无实根。当确定在$[x1,x2]$内有一个实根后，便采用二分法将$[x1,x2]$分成两部分，再确定实根在哪一个小区间中，依次类推，不断进行下去，直到分成的区间足够小为止。下面用二分法解方程 $\sin(x)=0$ 在（1.5,4.5）之间的根，区间范围由用户从键盘中输入。

用二分法解方程的算法如下。

（1）首先输入区间的范围$[x1,x2]$。

（2）求出 $f(x1)$及 $f(x2)$的值并判断是否异号，如果同号则说明在该区间内无实根，输出提示信息结束程序，否则继续进行步骤（3）。

（3）求 $x1$ 和 $x2$ 的中点 $x0$ 并求出 $f(x0)$的值。

（4）判断 $f(x0)$与 $f(x1)$是否同号，如果同号则根应在$[x0,x2]$中，否则说明根应在$[x1,x0]$中。

（5）判断 $f(x0)$的绝对值是否小于指定的精度，若不小于则跳转到步骤（3），按照上面的流程

继续执行，知道$f(x0)$的绝对值小于指定的精度，此时输出 x0 的值，该值便是所要求的近似根。

提示： 程序使用 do…while 语句判断输入的区间内是否有根，若无根则需重新输入范围，若有根则采用了二分法求解方程的根。

例 12.18 二分法求解方程（光盘位置：光盘\MR\Instance\12\18\18.c），该实例的主要代码如下：

```
#include <math.h>
#include <stdio.h>
main()
{
    float x0, x1, x2, f0, f1, f2;                    /*定义变量为浮点型*/
    do
    {
        printf("enter x1,x2:");
        scanf("%f,%f", &x1, &x2);                    /*从键盘中输入范围*/
        f1 = sin(x1);
        f2 = sin(x2);
    }
    while (f1 *f2 > 0);                              /*判断是否有根，若无根重新输入范围*/
    do
    {
        x0 = (x1 + x2) / 2;                          /*取中间值*/
        f0 = sin(x0);                                /*求 f0*/
        if ((f0 *f1) < 0)                            /*判断根在那个范围内*/
        {
            x2 = x0;                                 /*范围在（x1，x0）*/
            f2 = f0;
        }
        else
        {
            x1 = x0;                                 /*范围在（x0，x1）*/
            f1 = f0;
        }
    }
    while (fabs(f0) >= 1e-5);                        /*判断是否大于精度*/
    printf("x=%6.2f\n", x0);                         /*将最终结果输出*/
}
```

程序运行结果如图 12.18 所示。

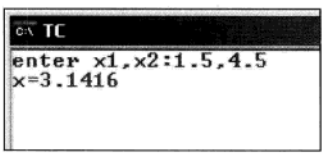


图 12.18 二分法求解方程

技巧 19 牛顿迭代法解方程的根

牛顿迭代法又称牛顿切线法，它的求根方法如下：先任意设定一个与真实的根接近的值 $x0$ 作为第一次近似根，由 $x0$ 求出 $f(x0)$，过$(x0, f(x0))$点做 $f(x)$的切线，交 x 轴于 $x1$，把它作为第二次近

似根；再由 $x1$ 求出求出 $f(x1)$，过$(x1, f(x1))$点做 $f(x)$的切线，交 x 轴于 $x2$，求出 $f(x2)$；再做切线……直到极为接近真正的根 xn 为止，因此得出牛顿迭代公式：$x1=x0-f(x0)/f'(x0)$。

采用牛顿迭代法求根。例如，方程为 $ax3+bx2+cx+d=0$，系数 a、b、c、d 的值从键盘中输入，求 x 在 1.5 附近的一个实根。

例 12.19　牛顿迭代法解方程的根（光盘位置：光盘\MR\Instance\12\19\ 19.c），该实例的主要代码如下：

```
//自定义 Newton 函数，实现迭代法求方程的根
float Newton(float a, float b, float c, float d)
{
    float x = 1, x0, f, f1;                          /*定义变量类型为单精度型*/
    do
    {
        x0 = x;
        f = ((a *x0 + b) *x0 + c) *x0 + d;           /*求 f(x0)*/
        f1 = (3 *a * x0 + 2 * b) *x0 + c;            /*求 f(x0)的导数*/
        x = x0 - f / f1;                             /*套用牛顿迭代公式*/
    }
    while (fabs(x - x0) >= 1e-5);                    /*迭代到指定精度*/
    return (x);                                      /*返回 x 的值*/
}
main()
{
    float a, b, c, d;
    scanf("%f%f%f%f", &a, &b, &c, &d);               /*从键盘中输入系数*/
    printf("%.2fx^3+%.2fx^2+%.2fx+%.2f=0\n", a, b, c, d);  /*输出方程*/
    printf("x=%.5f\n", Newton(a, b, c, d));          /*输出根*/
}
```

程序运行结果如图 12.19 所示。

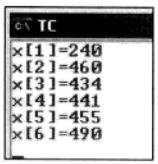


图 12.19　使用牛顿迭代法解方程的根

技巧 20　打印特殊方阵

打印特殊方阵，方阵具体要求如下：打印 9×9 阶的方阵，要求最外圈均是 1，次外圈均是 2，照此规律方阵最中间数字是 5。

分析方阵后发现该方阵上下左右是相互对称的，只要找出最外圈的是按什么规律构造的，也便找出了整个方阵的构造的规律，下面以最外圈为例分析，二维数组 a[i][j]其中 i 为 1 即行为 1，j 也代表了列从 1 变换到 9，这表示了整个方阵的最上面一行；a[10-i][j]中行为 9，列从 1 变化到 9，这表示了整个方阵的最下面一行；a[j][i]中行从 1 变化到 9，列为 1，则表示了整个方阵的最左面一列；

a[j][10-i]中行从 1 变化到 9，列为 9，这表示了整个方阵的最右面一列。按此规律，整个方阵便可构造出来。

例 12.20 打印特殊方阵（光盘位置：光盘\MR\Instance\12\20\20.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int a[10][10], i, j;                                    /*定义数组及变量为基本整型*/
    for (i = 1; i <= 5; i++)
    for (j = i; j <= 10-i; j++)
    {
        a[i][j] = i;                                        /*方阵中每次相对的上面一行等于 i*/
        a[10-i][j] = i;                                     /*方阵中每次相对的下面一行等于 i*/
        a[j][i] = i;                                        /*方阵中每次相对的左面一列等于 i*/
        a[j][10-i] = i;                                     /*方阵中每次相对的右面一列等于 i*/
    }
    for (i = 1; i <= 9; i++)
    {
        for (j = 1; j <= 9; j++)
            printf("%3d", a[i][j]);                         /*将该方阵输出*/
        printf("\n");                                       /*每输完一行进行换行*/
    }
}
```

程序运行的结果如图 12.20 所示。

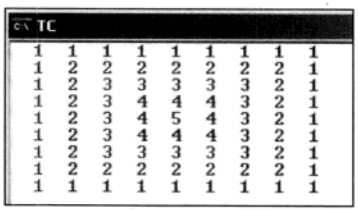


图 12.20 打印特殊方阵

技巧 21 求 3×3 矩阵对角元素之和

在矩阵运算中，经常会用到矩阵对角元素的和，通过键盘由用户任意输入 9 个数，组成一个 3×3 矩阵，然后求该矩阵的主对角线之和。

求矩阵对角元素之和关键是找出主对角线上各元素有什么特点，如果存储这个 3×3 矩阵的二维数组的下标从 1 行 1 列开始，可以发现主对角线上各元素的行数与列数是相等的，得出这一结论后就很容易求出主对角线上各元素之和了。这里有一点值得注意就是数组下标从 1 行 1 列开始使用的，故在定义二维数组时要定义成 a[4][4]。

使用嵌套的 for 循环将输入的数据存到二维数组 a 中，因为主对角线上的元素行列是相等的，故对满足该条件的元素进行累加求和。

例 12.21 求 3×3 矩阵对角元素之和（光盘位置：光盘\MR\Instance\12\21\ 21.c），该实例的主

要代码如下：

```
#include <stdio.h>
main()
{
    int a[4][4], i, j, sum = 0;
    printf("please input numbers:\n");
    for (i = 1; i <= 3; i++)
        for (j = 1; j <= 3; j++)
            scanf("%d", &a[i][j]);                    /*从键盘中输入 9 个数到二维数组 a*/
    printf("\nThis 3*3 matrix is:\n");
    for (i = 1; i <= 3; i++)
    {
        for (j = 1; j <= 3; j++)
            printf("%4d", a[i][j]);                   /*将 3×3 矩阵输出*/
        printf("\n");                                 /*每输出一行进行换行*/
    }
    for (i = 1; i <= 3; i++)
        for (j = 1; j <= 3; j++)
            if (i == j)                               /*如果函数等于列数，这说明是主对角线上的元素*/
                sum += a[i][j];                       /*对主对角线上的元素累加求和*/
    printf("diagonal's sum is %d", sum);              /*将最终结果输出*/
}
```

程序运行结果如图 12.21 所示。

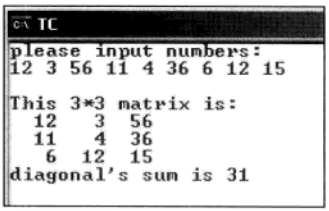


图 12.21　求 3×3 矩阵对角元素之和

技巧 22　矩阵的加法运算

矩阵是数的集合，它可以按照一定的规律进行加法运算。矩阵元素由键盘输入中获取，分别构造一个 2×3 的矩阵 *a* 和一个 3×2 的矩阵 *b*，然后求得矩阵 *a* 和矩阵 *b* 的加法运算结果，将运算结果保存在矩阵 *c* 中，将矩阵 *c* 输出到屏幕。

如果矩阵 *a* 和 *b* 分别是 3 行 3 列的，那么它们相加所得到的和即矩阵 *c* 也将是 3 行 3 列的，设 *a*、*b*、*c*3 个矩阵的第 *i* 行第 *j* 列元素分别为 *aij*、*bij*、*cij*，则 *cij* 的计算公式为：

$$cij=aij+bij,\qquad i=1,\ 2,\ 3,\ j=1,\ 2,\ 3$$

将上述计算公式中的 *i*、*j* 换成二维数组中的行列，便可编程实现求两个矩阵相加。

例 12.22　矩阵的加法运算（光盘位置：光盘\MR\Instance\12\22\22.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
```

```
{
    int a[3][3], b[3][3], c[3][3], i, j;
    printf("please input matrix a:\n");
    for (i = 0; i <= 2; i++)
        for (j = 0; j <= 2; j++)
            scanf("%d", &a[i][j]);                    /*从键盘中输入 3×3 的矩阵存入数组 a 中*/
    printf("please input matrix b:\n");
    for (i = 0; i <= 2; i++)
        for (j = 0; j <= 2; j++)
            scanf("%d", &b[i][j]);                    /*从键盘中输入 3×3 的矩阵存入数组 b 中*/
    for (i = 0; i <= 2; i++)
        for (j = 0; j <= 2; j++)
            c[i][j] = a[i][j] + b[i][j];              /*矩阵 a 和矩阵 b 中对应元素进行相加，和放入 c 中*/
    printf("the sum is:\n");
    for (i = 0; i <= 2; i++)
    {
        for (j = 0; j <= 2; j++)
            printf("%3d", c[i][j]);                   /*输出矩阵 c*/
        printf("\n");                                 /*每输出一行进行换行*/
    }
}
```

程序运行的结果如图 12.22 所示。

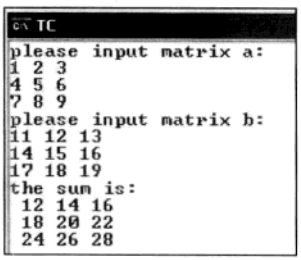


图 12.22　矩阵的加法运算

技巧 23　矩阵的乘法运算

矩阵是数的集合，它可以按照一定的规律进行乘法运算。矩阵元素由键盘输入中获取，分别构造一个 2×3 的矩阵 *a* 和一个 3×2 的矩阵 *b*，然后求得矩阵 *a* 和矩阵 *b* 的乘法运算结果，将运算结果保存在矩阵 *c* 中，将矩阵 *c* 输出到屏幕。

如果矩阵 *a* 和 *b* 分别是 2 行 3 列和 3 行 2 列的，那么它们相乘所得到的积即矩阵 *c* 将是 2 行 2 列的，设 *cik* 是矩阵 *c* 中 *i* 行 *k* 列的元素，*aij* 和 *bjk*（*j*=1,2,3,*i*=1,2,*k*=1,2）分别表示矩阵 *a* 的第 *i* 行第 *j* 列元素和矩阵 *b* 的第 *j* 行第 *k* 列元素，那么就有 *cik*=*ai*1×*b*1*k*+*ai*2×*b*2*k*+*ai*3×*b*3*k*。将计算公式中的 *i*、*j*、*k* 换成二维数组中的行列，便可实现两个矩阵相乘。

例 12.23　矩阵的乘法运算（光盘位置：光盘\MR\Instance\12\23\23.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int a[2][3], b[3][2], c[2][2], i, j, i1, sum;
```

```
    printf("please input martrix a:\n");
    for (i = 0; i <= 1; i++)
        for (j = 0; j <= 2; j++)
            scanf("%d", &a[i][j]);                    /*输入 2*3 矩阵到数组 a 中*/
    printf("please input martrix b:\n");
    for (i = 0; i <= 2; i++)
        for (j = 0; j <= 1; j++)
            scanf("%d", &b[i][j]);                    /*输入 3*2 矩阵到数组 b 中*/
    for (i = 0; i <= 1; i++)                          /*a 矩阵的行数*/
        for (k = 0; k <= 1; k++)                      /*b 矩阵的列数*/
    {
        sum = 0;
        for (j = 0; j <= 2; j++)
                sum += a[i][j] *b[j][k];
        c[i][k] = sum;                                /*每次求得的和放入指定的行列中*/
    }
    printf("the product is:\n");
    for (i = 0; i <= 1; i++)
    {
        for (j = 0; j <= 1; j++)
            printf("%4d", c[i][j]);                   /*将矩阵 c 输出*/
        printf("\n");                                 /*每输出一行进行换行*/
    }
}
```

程序运行的结果如图 12.23 所示。

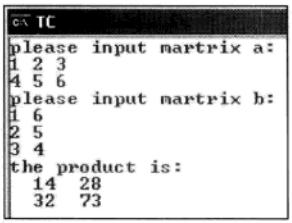


图 12.23　矩阵的乘法运算

技巧 24　打印 n 阶螺旋方阵

螺旋方阵是一种特殊的矩阵，可以从键盘中任意输入一个数 n（n<=10），然后在屏幕中打印出 n 阶螺旋方阵。打印出 n 阶螺旋方阵的过程主要是使用嵌套的 for 循环将变量 k 的值分别存入每圈数组元素中，上面 a[i][j]、右面 a[j][n-i-1]、下面 a[n - i - 1][j]及左面 a[j][i]，然后将二维数组正常输出便可以得到螺旋方阵。

提示： 输出螺旋方阵时具体的输出过程并不难，但是在将数据往数组存储的过程中要注意存储的顺序即：上→左→下→右，因为程序中始终是将变量 k 的值向数组中存储，若不注意顺序输出时便会产生错误。

例 12.24　打印 n 阶螺旋方阵（光盘位置：光盘\MR\Instance\12\24\24.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int i, j, k = 0, a[10][10], m, n;                    /*定义变量及二维数组 a 分别为基本整型*/
    printf("please input n:\n");
    scanf("%d", &n);
    if (n % 2 == 0)                                      /*判断是否为偶数*/
        m = n / 2;
    else
        m = n / 2+1;
    for (i = 0; i < m; i++)
    {
        for (j = i; j < n - i; j++)
        {
            k++;
            a[i][j] = k;                                 /*将每圈中上面的行中的数据存入数组中*/
        }
        for (j = i + 1; j < n - i; j++)
        {
            k++;
            a[j][n - i - 1] = k;                         /*将每圈中右面的列中的数据存入数组中*/
        }
        for (j = n - i - 2; j >= i; j--)
        {
            k++;
            a[n - i - 1][j] = k;                         /*将每圈中下面行中的数据存入数组中*/
        }
        for (j = n - i - 2; j >= i + 1; j--)
        {
            k++;
            a[j][i] = k;                                 /*将每圈中左面的列中的数据存入数组中*/
        }
    }
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            printf("%5d", a[i][j]);                      /*将螺旋矩阵输出*/
        printf("\n");                                    /*每输出一行进行换行操作*/
    }
}
```

程序运行的结果如图 12.24 所示。

```
TC
please input n:
8
    1    2    3    4    5    6    7    8
   28   29   30   31   32   33   34    9
   27   48   49   50   51   52   35   10
   26   47   60   61   62   53   36   11
   25   46   59   64   63   54   37   12
   24   45   58   57   56   55   38   13
   23   44   43   42   41   40   39   14
   22   21   20   19   18   17   16   15
```

图 12.24　打印 *n* 阶螺旋方阵

技巧 25　求车运行速度

一辆以固定速度行驶的汽车，司机在上午 8 点看到里程表上的读数是一个对称数（即这个数从

左向右读和从右向左读是完全一样的），为 95859。两小时后里程表上出现了一个新的对称数。问该车的速度是多少？新的对称数是多少？

该问题的关键是如何进行对称数的判断。将探测到的 5 位数的各位分解存到数组中，然后根据数组对称位置元素的比较，就可以判断一个数是否为对称数，可以在 while 循环中实现该功能。例如，如果有整型数 i 和数组 a，i=95886，当 k=100000 时，a[1]=95886%100000/10000，此时 a[1]中存放的数字是 9，k 通过 k/=10 被重新赋值为 10000，第二次循环 a[2]=95886%10000/1000，此时 a[2]中存放的是 5，依次类推，a[3]、a[4]、a[5]中存放的分别是 8、8、6，这样就可以通过比较第 1 位和第 5 位、第 2 位和第 4 位是否相同来判断该数字是否是我们要找的对称数。

提示：程序中使用嵌套的 while 循环，外层循环的作用是从 95860 开始的整数进行探测，直到找到对称数为止，内层循环的作用是将探测到的 5 位数逐位分解存到数组 a 中。

例 12.25　求车运行速度（光盘位置：光盘\MR\Instance\12\25\25.c），该实例的主要代码如下：

```
#include<stdio.h>
main()
{
    int j=0,a[5];                                    /*数组 a 存放分解的数字位*/
    long k=100000,i=95860;                           /*因为 95860 是 5 位数故 k=100000*/
    while(1)
    {      j=0;/*j 为数组下标*/
    k=100000;
    while(k>=10)
        {
        a[j]=(i%k)/(k/10);                           /*从高到低分解 i 值的每位数存到数组中*/
        k/=10;
        j++;
        }
        if((a[0]==a[4])&&(a[1]==a[3]))               /*比较第 1 位和第 5 位、第 2 位和第 4 位是否相同*/
        {
        printf("The new number kelometers is:%d%d%d%d%d\n",
                                  a[0],a[1],a[2],a[3],a[4]);    /*输出新的对称数*/
        printf("an average speed is: %.2f\n",(i-95859)/2.0);    /*将求出的平均速度输出*/
            break;
        }
         i++;
    }
}
```

程序运行的结果如图 12.25 所示。

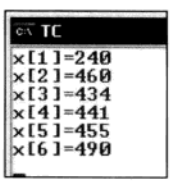


图 12.25　求车运行速度

技巧 26 卖西瓜

一农户在集市上卖西瓜，总共有 1020 个西瓜，第一天卖掉一半多两个，第二天卖掉剩下的一半多两个，问照此规律买下去，该农户几天能将所有的西瓜买完？

分析该问题，假设 x1 为西瓜的总数，每天西瓜的总数都按同一规律变化，即前一天西瓜数的一半再减 2，用 while 循环来实现每天的变化，那么循环结束条件是什么呢？这里采用西瓜的总数作为循环结束的条件，当西瓜总数为 0 时，循环结束。

例 12.26 卖西瓜（光盘位置：光盘\MR\Instance\12\26\26.c），该实例的主要代码如下：

```
#include<stdio.h>
main()
{
    int day, x1, x2;
    day = 0;                                    /*天数*/
    x1 = 1020;                                  /*瓜的总数量*/
    while (x1)
    {
        x2 = x1 / 2-2;                          /*卖出一半多两个*/
        x1 = x2;
        day++;                                  /*天数加 1*/
    }
    printf("day=%d\n", day);                    /*将天数加 1*/
}
```

程序运行的结果如图 12.26 所示。

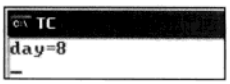


图 12.26 卖西瓜

技巧 27 巧分苹果

一家农户以果园为生，一天，父亲推出一车苹果，共 2520 个，准备分给他的 6 个儿子。父亲先按事先写在一张纸上的数字把这堆苹果分完，每个人分到的苹果个数都不相同。然后他说：“老大，把你分到的苹果分 1/8 给老二，老二拿到后，连同原来的苹果分 1/7 给老三，老三拿到后，连同原来的苹果分 1/6 给老四，依次类推，最后老六拿到后，连同原来的苹果分 1/3 给老大，这样，你们每个人分到的苹果就一样多了。”问兄弟 6 人原先各分到多少只苹果？

要解决本实例首先要分析其中的规律，这里我们设 xi（i=1,2,3,4,5,6）依次为 6 个兄弟原来分到的苹果数，设 yi=（2,3,4,5,6）为出老大外其余 5 个兄弟从哥哥那里得到还未分给弟弟时的苹果数，那么老大是个特例则 x1=y1。因为苹果的总数是 2520，那么我们可以很容易就知道 6 个人平均每人得到的苹果数 s 应为 420，则可得到如下关系：

y2=x2+(1/8)*y1，

y2*(6/7)=s；

y3=x3+(1/7)*y2，

y3*(5/6)=s；

y4=x4+(1/6)*y3，

y4*(4/5)=s；

y5=x5+(1/5)*y4，

y5*(3/4)=s；

y6=x6+(1/4)*y5，

y6*(2/3)=s；

以上求 s 都是有规律的，对于老大的求法这里单列，即 y1=x1，x1*(7/8)+y6*(1/3)=s。

根据上面分析的内容，我们利用数组便可实现巧分苹果：首先利用循环和数组先求出从哥哥得到分来的苹果却未分给弟弟时的数目，在该数的基础上再求原来每人分到的苹果数。

例 12.27　巧分苹果（光盘位置：光盘\MR\Instance\12\27\27.c），该实例的主要代码如下：

```
#include <stdio.h>
main()
{
    int x[7], y[7], s, i;
    s = 2520 / 6;                                          /*求出平均每个人要分多少个苹果*/
    for (i = 2; i <= 6; i++)
     /*求从老二到老六得到哥哥分来的苹果却未分给弟弟时的苹果数*/
        y[i] = s *(9-i) / (8-i);
    y[1] = x[1] = (s - y[6] / 3) *8 / 7;
     /*老大得到老六分来的苹果却未分给弟弟时的苹果数*/
    for (i = 2; i <= 6; i++)
        x[i] = y[i] - y[i - 1] / (10-i);                   /*求原来每人得到的苹果数*/
    for (i = 1; i <= 6; i++)
        printf("x[%d]=%d\n", i, x[i]);                     /*将最终结果输出*/
}
```

程序运行的结果如图 12.27 所示。

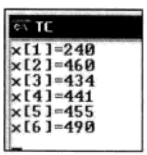


图 12.27　巧分苹果



第13章 C语言开发人员必知必会的数据结构开发应用技巧

数据结构在 C 语言程序开发中经常使用，是开发人员必须熟练掌握的内容之一。本章对数据结构中那些经常使用但又不易掌握的数据结构开发技巧进行了讲解，这些技巧包括了结构体、链表、栈和队列、串与广义表等，通过学习本章，读者可以掌握数据结构开发中的应用技巧，为开发数据结构 C 语言程序打下坚实的基础。

技巧 01 结构体简单应用

本技巧创建一个名为 student 的结构体类型，其中包含了学生的基本信息。定义一个 student 类型的变量，并使用学生信息为该变量赋值。运行程序后，在窗体上输出该变量的各成员的值，即将学生信息进行输入。

例 13.1 创建一个名为 student 的结构体类型，其中包含了学生的基本信息，在 main()主函数中为结构体类型的变量赋值，并在窗口中输出结构体变量的值(光盘位置：光盘\MR\Instance\13\01)。

（1）在 TC 环境中创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>
```

（3）定义一个结构体类型。

```
struct student
{
    /*声明结构体成员*/
    int num;
    char name[20];
    char sex;
    int age;
    float score;
};
```

（4）创建 main()函数作为程序的入口程序，在 main()函数中定义 struct student 类型的变量。为结构体类型的变量赋值，并在窗口中输出结构体变量的值。具体程序代码如下：

```
void main()
{
    struct student student1={1001,"liming",'M',20,93.5};          /*定义结构体变量并初始化*/
    printf("The information of the student is:\n");
    printf("num:%d\n",student1.num);                              /*输出学生学号*/
    printf("name:%s\n",student1.name);                            /*输出学生姓名*/
    printf("sex:%c\n",student1.sex);                              /*输出学生性别*/
    printf("age:%d\n",student1.age);                              /*输出学生年龄*/
    printf("score:%5.1f\n",student1.score);                       /*输出学生成绩*/
    getch();
}
```

运行本实例，效果如图 13.1 所示。

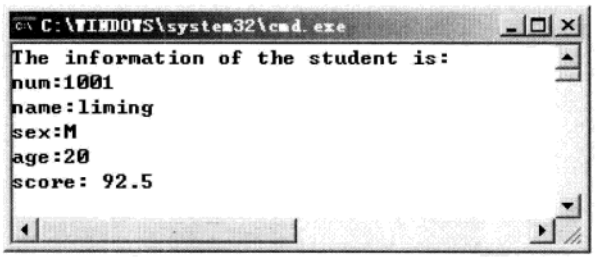


图 13.1　结构体简单应用

技巧解析：

结构体是一种构造类型，像数组一样结构体也是由若干元素组成。但是结构体中的元素被称为“成员”，并且数组中的元素要具有相同的数据类型，而结构体的成员是包含若干个类型（当然也可以相同）的数据项。

例如，学生信息记录要包括学生的姓名、性别、年龄、学号、成绩等信息，而姓名应为字符型，学号应为整型或是字符型，年龄应为整型。显然不能使用数组来存放这些数据，而将这些数据分别定义为相互独立的简单变量又难以反映它们之间的内在联系。

这时使用结构体类型就比较方便了。下面介绍结构体类型的相关知识。

- 指定结构体类型。

结构体是用户自定义的一种构造的数据类型。因此在使用结构体之前要先进行定义。定义的一般形式为：

```
struct 结构体名
   {成员表列}
```

Struct：是声明结构体类型时所必须使用的关键字，不能省略。它向编译系统说明这是一个“结构体类型”。

结构体名：是结构体类型的标志。

成员表列：是该结构体中的各个成员，由它们组成一个结构体。对各成员都应进行类型声明，其形式为：

```
类型名  成员名
```

成员名命名规则与变量名相同。例如，本程序中定义了结构体 student，其定义语句如下：

```
struct student
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
};
```

该结构体有 5 个成员，两个整型变量 num 和 age，一个字符串数组 name,一个字符型变量 sex 和一个浮点型变量 score。这些成员组合到一起就可以表示一个学生的记录了。

注意：定义的一个结构体只是一个类型，同整型（int）、字符型（char）、浮点型（float）一样，而并非一个实体变量，只不过这个类型是用户自己定义的构造类型，而不是系统定义的类型。

- 定义结构体类型变量。

定义一个结构体类型是为了在程序中使用这个结构体类型，为了能在程序中使用结构体类型的数据，应当定义结构体类型的变量，并为该变量赋值。下面介绍几种定义结构体类型变量的方法。

- 先声明结构体类型再定义变量。

先声明结构体类型再定义变量。例如下面的语句：

```
struct student
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
};
struct student stu1,stu2;
```

先定义结构体类型 struct student，再定义两个结构体变量 stu1 和 stu2。stu1 和 stu2 即是结构体变量名。

定义了结构体变量，系统就要为该变量分配内存空间。系统为结构体变量分配的空间大小与定义的结构体类型有关。例如，上面定义的结构体变量包含了 5 个成员，根据每个成员的类型可以得到，该结构体类型变量 stu1 和 stu2 的存储空间都为 29（2+20+1+2+4=29）字节。

- 声明结构体类型的同时定义变量。

例如：

```
struct student
{
    int num;
    char name[20];
    char sex;
```

```
    int age;
    float score;
}stu1,stu2;
```

采用这种方法定义的结构体变量为全局变量。

- 直接定义结构体类型变量。

也可以直接定义结构体变量，而声明部分不需要给出结构体名称，例如：

```
struct
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
}stu1,stu2;
```

直接定义结构体变量 stu1 和 stu2，而不需要出现结构体名。

上面 3 种定义结构体变量的方法所定义的变量是本质相同的变量，只是定义时的写法不同而已。定义好的结构体变量就可以为其赋值了。

- 结构体变量的引用。

引用结构体变量中成员的方式为：

```
结构体变量名.成员名
```

例如，引用 stu1 的 name 成员，可写作 stu1.name。也可以使用此种方式为该变量的成员赋值，例如：

```
stu1.name= "Jam"
```

不能将一个结构体变量作为一个整体进行输入或是输出。例如，对于 stu1 下面的输出方法是不正确的：

```
printf("%d,%s,%c,%d,%f\n",stu1);
```

技巧 02　找最高分

本技巧通过结构体变量记录学生成绩，比较得到学生成绩记录中的最高成绩，并将该学生的信息输出。

例 13.2　创建一个名为 struct student 的结构体，在 main()主函数中实现查找数组中的最高分，并显示在窗体上，实现步骤如下（光盘位置：光盘\MR\Instance\13\02）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include "stdio.h"
```

（3）声明 struct student 类型。

```
struct student
{
    /*结构体成员*/
    int num;
    char name[20];
    float score;
};
```

（4）创建 main()函数作为程序的入口程序，在 main()函数中定义 struct student 类型的数组。查找数组中各学生记录的最高分，并显示在窗体上。实现代码如下：

```
void main()
{
    int i, m;                                          /*定义变量*/
    float maxscore;
    struct student stu[5] =                            /*为数组赋值*/
    {
        {101, "liming", 89} ,
        {102, "zhanghong", 95},
        {103, "lili", 89},
        {104, "weichen", 85},
        {105, "yangfan", 75}
    };                                                 /*声明结构体类型数组*/
    m = 0;
    maxscore = stu[0].score;                           /*初始化最大成绩*/
    for (i=1;i<5;i++)                                  /*i 的取值范围为 1~4*/
    {
        if (stu[i].score > maxscore)
        {
            maxscore = stu[i].score;                   /*记录最大成绩*/
            m = i;                                     /*记录最大成绩下标*/
        }
    }
    printf("The maxmum score is :%5.1f\n", maxscore);  /*输出最大成绩*/
    printf("The student number is: %d\n", stu[m].num); /*最大成绩的学号*/
    printf("The student name is: %s\n", stu[m].name);  /*最大成绩的姓名*/
    getch();
}
```

运行本实例，效果如图 13.2 所示。

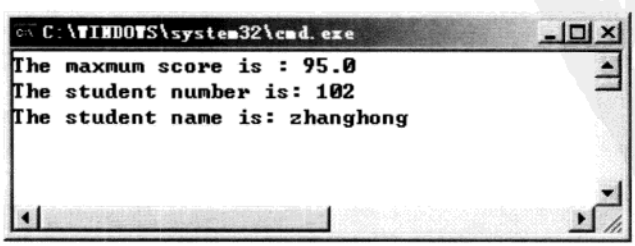


图 13.2　显示最高分学生信息

技巧 03　平均成绩

利用结构体，编程实现输入一个学生期中和期末成绩求出其平均成绩。

实现思想：

本技巧是一个简单的结构体应用，没有太多难点，定义的结构体成员变量包括期中、期末及平均成绩。从键盘中输入期中、期末成绩存到定义的结构体 score 中，计算出平均成绩即可。

例 13.3　定义一个名称为 score 的结构体，将期中和期末的成绩存到结构体 score 中，将期中和期末的成绩求和并除以 2，求出平均成绩，将结果输出在窗口上（光盘位置：光盘\MR\Instance\13\03）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>
```

（3）输入期中、期末成绩存到定义的结构体 score 中，将两个成绩求和再除以 2 便可求出平均成绩。

（4）主函数程序代码。

```
main()
{
    struct student_score                                    /*定义结构体，存储期中、期末及平均成绩
*/
    {
        int mid;                                            /*定义变量*/
        int end;
        int ave;
    }score;
        printf("please input score(midterm and end of term):"); /*输出提示信息*/
        scanf("%d,%d",&score.mid,&score.end);               /*输入期中、期末成绩*/
    score.ave = (score.mid + score.end)/2;                  /*计算出平均成绩*/
    printf("average=%d\n", score.ave);                      /*将平均成绩输出*/
}
```

运行本实例，效果如图 13.3 所示。

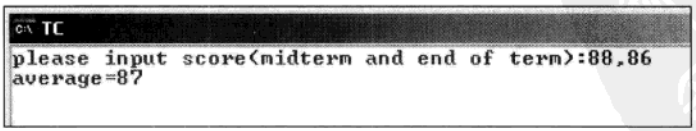


图 13.3　平均成绩

技巧 04　比较计数

用“比较计数”法对结构数组按字段 num 进行升序排序，num 的值从键盘中输入。

实现思想：

定义结构体用来存储输入的数据及其最后排序，对数组中的元素逐个比较并用结构体中成员变量 con 记录该元素大于其他元素的次数，次数越大则证明该数据越大。

例 13.4 定义结构体 order 用来存储数据及其排序，并定义结构体数组 a，对数组中的每个元素和其他元素比较，最后将排序后的数组输出在窗体显示。实现步骤如下（光盘位置：光盘\MR\Instance\13\04）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，进行宏定义。

```
#include<stdio.h>
#define N 5
```

（3）定义结构体 order 用来存储数据及其排序，并定义结构体数组 a，代码如下：

```
struct order                                    /*定义结构体用来存储数据及它的排序*/
{
    int num;
    int con;
} a[20];                                        /*定义结构体数组 a*/
```

（4）主函数程序代码。

```
main()
{
    int i, j;
    for (i = 0; i < N; i++)
    {
        scanf("%d", &a[i].num);                 /*输入要进行排序的 5 个数字*/
        a[i].con = 0;
    }
    for (i = N - 1; i >= 1; i--)
        for (j = i - 1; j >= 0; j--)
            if (a[i].num < a[j].num)            /*对数组中的每个元素和其他元素比较*/
                a[j].con++;                     /*记录排序号*/
    else
        a[i].con++;
    printf("the order is:\n")for (i = 0; i < N; i++)
        printf("%3d%3d\n", a[i].num, a[i].con); /*将数据及其排序输出*/
}
```

运行本实例，效果如图 13.4 所示。

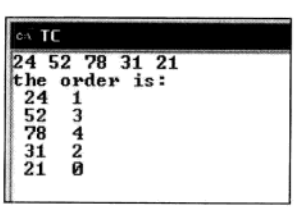


图 13.4 比较计数

技巧 05　信息查询

在信息时代的今天，当需要查找某个重要的信息时，面对众多信息常常弄的焦头烂额。本技巧从键盘中输入姓名和电话号码，以#结束，编程实现输入姓名、查询电话号码的功能。

实现思想：

首先定义一个结构体用来存储姓名及电话号码，再分别定义两个函数，一个函数的作用是将输出的姓名及电话号码存储到结构体数组中，另一个函数的功能是根据输入的姓名查找电话号码，最后在主函数中分别调用这两个函数就能实现以上要求的功能。

例 13.5　定义一个结构体和两个自定义函数，结构体用来储存电话号码和姓名，其中一个自定义函数用来实现电话号码和姓名存储的过程，另一个自定函数用来查询输入的姓名所对应的电话号码，实现步骤如下（光盘位置：光盘\MR\Instance\13\05）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件并进行宏定义。

```
#include <stdio.h>
#include <string.h>
#define MAX 101
```

（3）定义结构体 aa，用来储存电话号码和姓名。

```
struct aa                                          /*定义结构体 aa 存储姓名和电话号码*/
{
char name[15];
char tel[15];
};
```

（4）自定义函数 readin()，用来实现电话号码和姓名存储的过程，代码如下：

```
int readin(struct aa *a)                           /*自定义函数 readin，用来存储姓名及电话号码*/
{
    int i = 0, n = 0;
    while (1)
    {
        scanf("%s", a[i].name);                    /*输入姓名*/
        if (!strcmp(a[i].name, "#"))
            break;
        scanf("%s", a[i].tel);                     /*输入电话号码*/
        i++;
        n++;                                       /*记录的条数*/
    }
    return n;                                      /*返回条数*/
}
```

（5）自定义函数 search()，用来查询输入的姓名所对应的电话号码，代码如下：

```
void search(struct aa *b, char *x, int n)          /*自定义函数 search，查找姓名所对应的电话号码*/
{
```

```
    int i;
    i = 0;
    while (1)
    {
        if (!strcmp(b[i].name, x))                          /*查找与输入姓名相匹配的记录*/
        {
            printf("name:%s  tel:%s\n", b[i].name, b[i].tel);   /*输出查找到的姓名所对应的电话号码*/
            break;
        }
        else
            i++;
        n--;
        if (n == 0)
        {
            printf("No found!");                            /*若没查找到记录输出提示信息*/
            break;
        }
    }
}
```

（6）主函数程序代码。

```
main()
{
    struct aa s[MAX];                                       /*定义结构体数组 s*/
    int num;
    char name[15];
    num = readin(s);                                        /*调用函数 readin*/
    printf("input the name:");
    scanf("%s", name);                                      /*输入要查找的姓名*/
    search(s, name, num);                                   /*调用函数 search*/
}
```

运行本实例，效果如图 13.5 所示。

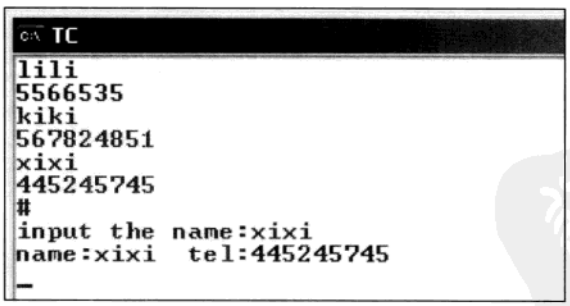


图 13.5　信息查询

技巧 06　计算开机时间

编程实现计算开机时间，要求在每次开始计算开机时间时都能接着上次记录的结果继续向下记录。

实现思想：

实例中按秒为单位读取系统时间，将读取的时间存到指定磁盘文件中，每次开始计时的时候就从该磁盘文件中读取上次记录的时间接着计时，当秒数达到 60，则分钟数加 1，如果分钟数达到 60，则小时数加 1。

例 13.6　定义结构体 time，用来存储时间信息，在主函数中实现计算开机时间，实现步骤如下所示（光盘位置：光盘\MR\Instance\13\06）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）定义结构体 time，用来存储时间信息，代码如下：

```
struct time                                          /*定义结构体 time，存储时间信息*/
{
    int hour;
    int minute;
    int second;
}t;
```

（4）主函数程序代码。

```
main()
{
    FILE *fp;                                        /*定义文件类型指针 fp*/
    fp = fopen("Time", "r");                         /*以只读方式打开文件 Time*/
    fread(&t, sizeof(struct time), 1, fp);           /*读取文件中信息*/
    while (!kbhit())                                 /*当无按键时执行循环体语句*/
    {
        rewind(fp);                                  /*将文件指针设置到文件起点*/
        sleep(1);                                    /*程序停止 1 秒钟*/
        fread(&t, sizeof(struct time), 1, fp);       /*读取文件中的内容*/
        if (t.second == 59)                          /*如果到 60 秒*/
        {
            t.minute = t.minute + 1;                 /*如果到 60 秒分钟数加 1*/
            if (t.minute == 60)                      /*判断是否到 60 分钟*/
            {
                t.hour = t.hour + 1;                 /*到 60 分钟小时数加 1*/
                t.minute = 0;                        /*分数置 0*/
            } t.second = 0;                          /*秒数置 0*/
        }
        else
            t.second = t.second + 1;                 /*秒数加 1*/
        printf("%d:%d:%d\n", t.hour, t.minute, t.second);  /*输出累积开机时间*/
        fp = fopen("Time", "w");                     /*以可写方式打开 Time 文件*/
        fwrite(&t, sizeof(struct time), 1, fp);      /*定义结构体 time，存储时间信息*/
        fclose(fp);                                  /*关闭文件指针*/
    }
}
```

运行本实例，效果如图 13.6 所示。

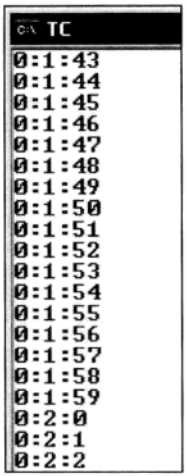


图 13.6　计算开机时间

技巧 07　创建双向链表

本技巧实现创建一个双向链表，并将这个链表中数据输出到窗体上，输入要查找的学生姓名，将查找的姓名从链表中删除，并显示删除后的链表。

实现思想：

单向链表节点的存储结构只有一个指向直接后继的指针域，所以，从单链表的某个节点出发只能顺指针查找其他节点。使用双向链表可以避免单链表这种单向性的缺点。

顾名思义，双向链表的节点有两个指针域，一个指向其直接后继，另一个指向其直接前驱，在 C 语言中可描述如下：

```
typedef struct DulNode
{
    char name[20];
    struct node *prior;                    /*直接前驱指针*/
    struct node *next;                     /*直接后继指针*/
}DNode;
```

其结构如图 13.7 所示。

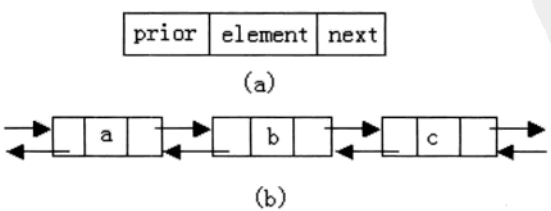


图 13.7　双向链表示意图

如图 13.7（a）所示，双向链表包括 3 个域，两个指针域一个数据域。如图 13.7（b）所示，可以看出双向链表节点间的关系。

例 13.7　声明名为 struct LNode 的结构体类型。创建 create()自定义函数，实现创建一个双向链表。创建 search()自定义函数，实现查找要删除的节点，如果找到则返回该节点地址。创建 del()自定义函数，实现删除链表中指定的节点。实现步骤如下所示（光盘位置：光盘\MR\Instance\13\07）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）声明 struct LNode 类型。

```
typedef struct node
{
    char name[20];
    struct node *prior,  *next;
} stud;                                          /*双链表的结构定义*/
```

（4）创建 create()自定义函数，实现创建一个双向链表，将此函数定义为指针类型，使其返回值为指针值，返回值指向一个 struct node 类型数据，实际上是返回链表的头指针。

```
stud *create(int n)
{
    stud *p,  *h,  *s;
    int i;
    h = (stud*)malloc(sizeof(stud));                    /*申请节点空间*/
    h->name[0]='\0';
    h->prior =NULL;
    h->next=NULL;
    p = h;
    for (i = 0; i < n; i++)
    {
        s = (stud*)malloc(sizeof(stud));
        p->next = s;                                    /*指定后继节点*/
        printf("Input the %d student's name: ", i + 1);
        scanf("%s", s->name);
        s->prior = p;                                   /*指定前驱节点*/
        s->next = NULL;
        p = s;
    }
    p->next = NULL;
    return (h);
}
```

创建 search()自定义函数，实现查找要删除的节点，如果找到则返回该节点地址。

```
/*查找*/
stud *search(stud *h, char *x)
{
    stud *p;                                        /*指向结构体类型的指针*/
    char *y;                                        /*定义字符型指针*/
    p = h->next;
    while (p)
```

```
    {
        y = p->name;
        if (strcmp(y, x) == 0)                          /*如果是要删除的节点，则返回地址*/
            return (p);
        else
            p = p->next;
    }
    printf("cannot find data!\n");
}
```

创建 del()自定义函数，实现删除链表中指定的节点。

```
/*删除*/
void del(stud *p)
{
    p->next->prior = p->prior;                  /*p 的下一个节点的前驱指针指向 p 的前驱节点*/
    p->prior->next = p->next;                   /*p 的前驱节点的后继指针指向 p 的后继节点*/
    free(p);
}
```

（4）创建 main()函数作为程序的入口程序，在 main()函数中调用 create()自定义函数，实现创建一个链表，并将链表中的数据输出。调用 search()自定义函数和 del()自定义函数实现查找指定节点并从链表中将该节点删除。实现代码如下：

```
/*主函数*/
main()
{
    int number;                                     /*定义变量*/
    char sname[20];                                 /*定义字符型数组*/
    stud *head,*sp;
    puts("Please input the size of the list:");
    scanf("%d", &number);                           /*输入链表节点数*/
    head = creat(number);                           /*创建链表*/
    sp = head->next;
    printf("\nNow the double list is:\n");
    while (sp)                                      /*输出链表中数据*/
    {
        printf("%s ", &*(sp->name));
        sp = sp->next;
    }
    printf("\nPlease input the name which you want to find:\n");
    scanf("%s", sname);
    sp = search(head, sname);                       /*查找指定节点*/
    printf("the name you want to find is:%s\n",  * &sp->name);
    del(sp);                                        /*删除节点*/
    sp = head->next;
    printf("\nNow the double list is:\n");
    while (sp)
    {
        printf("%s ", &*(sp->name));                /*输出当前链表中数据*/
        sp = sp->next;
    }
    printf("\n");                                   /*换行*/
    puts("\n Press any key to quit...");
    getch();
}
```

运行本实例，效果如图 13.8 所示。

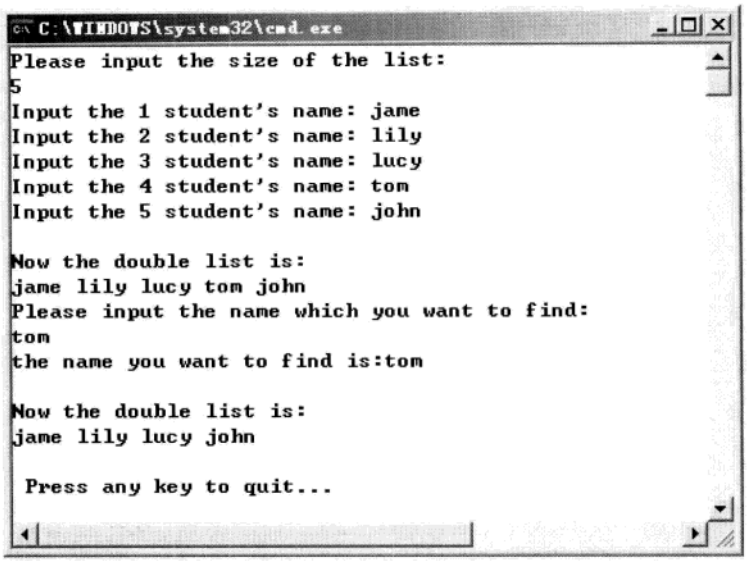


图 13.8　创建单向链表

技巧解析：

链表是动态分配存储空间的链式存储结构，其包括一个“头指针”变量，头指针变量中存放一个地址，该地址指向一个元素。链表中每一个元素称为“节点”，每个节点都由两部分组成：存储数据元素的数据域和存储直接后继存储位置的指针域。指针域中存储的即是链表的下一个节点的存储位置，是一个指针。多个节点结成一个链表。链表的结构如图 13.9 所示。

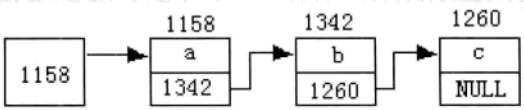


图 13.9　链表结构示意图

其中第 0 个节点称为整个链表的头节点，它一般不存放具体数据，只是存放第一个节点的地址，也称为“头指针”。最后一个节点的指针域设置为空（NULL），作为链表的结束标志，表示它没有后继节点。

说明：从图 13.9 可以看出，链表并不一定是连续存放的存储节点，并且只要获得链表的头节点，就可以通过指针变量整条链表。

使用结构体变量作为链表中的节点，因为结构体变量成员可以是数值类型、字符类型、数组类型，也可以是指针类型，这样就可以使用指针类型成员来存放下一个节点的地址，使用其他类型成员存放数据信息。例如，可以一个结构体类型如下：

```
struct LNode
{
```

```
    int data;
    struct LNode *next;
};
```

上面的结构体类型中成员 data 用来存放节点中的数据，相当于图 13.8 中的 a、b、c。next 是指针类型的成员，它指向 struct LNode 类型数据，就是本结构体类型的数据。

注意：上面只是定义了一个结构体类型，并未实际分配存储空间，只有定义了变量才非配存储空间。

在创建链表时要动态地为链表分配空间，C 语言的库函数提供了几种函数来实现动态开辟存储单元。这里使用 malloc()函数实现动态的开辟存储单元，下面进行介绍。

malloc 函数原型为：

```
void *malloc(unsigned int size);
```

其作用是在内存的动态存储区中分配一个长度为 size 的连续空间。函数返回值是一个指向分配域起始地址的指针（类型为 void）。如果分配空间失败（例如，内存空间不足），则返回空指针（NULL）。

技巧 08 创建循环链表

本技巧用于创建一个循环链表。这里只创建一个简单的循环链表来演示循环链表的创建和输出方法。

实现思想：

循环链表是另一种形式的链式存储结构。只是链表中最后一个节点的指针域指向头节点，使链表形成一个环。从表中任一节点出发均可找到表中其他节点，如图 13.10 所示为单链表的循环链表结构示意图。也可以有双链表的循环链表。

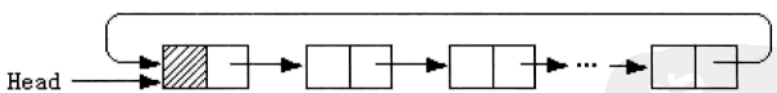


图 13.10 循环链表示意图

循环链表与普通链表的操作基本一致，只是在算法中循环遍历链表节点时判断条件不再是 p->next 是否为空，而是是否等于链表的头指针。

例 13.8 声明 struct student 类型，创建 create()自定义函数，实现创建一个循环链表，建 main()函数作为程序的入口函数，实现调用 create()自定义函数创建一个循环链表，并输出（光盘位置：光盘\MR\Instance\13\08）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）声明 struct student 类型。

```
typedef struct student
{
    int num;
    struct student *next;
} LNode,*LinkList;
```

（4）创建 create()自定义函数，实现创建一个循环链表，其返回值为指针值，返回值指向一个 struct node 类型数据，实际上是返回链表的头指针。

```
LinkList create(void)
{
    LinkList head;
    LNode *p1,*p2;
    char a;
    head = NULL;
    a = getchar();
    while (a != '\n')
    {
        p1 = (LNode*)malloc(sizeof(LNode));                    /*分配空间*/
        p1->num = a;                                           /*数据域赋值*/
        if (head == NULL)
            head = p1;
        else
            p2->next = p1;
        p2 = p1;
        a = getchar();
    }
    p2->next = head;                                           /*尾节点指向头节点*/
    return head;
}
```

（5）创建 main()函数作为程序的入口函数，实现调用 create()自定义函数创建一个循环链表，并输出。

```
void main()
{
    LinkList L1, head;
    printf("Please input the linklist:\n");                    /*输出提示信息*/
    L1 = create();                                             /*创建循环链表*/
    head = L1;
    printf("The resultant linklist is:\n");
    printf("%c ", L1->num);
    L1 = L1->next;                                             /*指向下一个节点*/
    while (L1 != head)
    {
        /*判断条件为循环到头节点结束*/
        printf("%c ", L1->num);
        L1 = L1->next;
    }
    getch();
}
```

运行本实例，效果如图 13.11 所示。

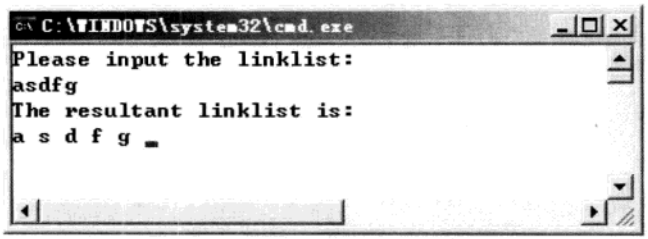


图 13.11　创建循环链表

技巧 09　创建顺序表并插入元素

本实例实现创建一个顺序表，在顺表中插入元素，并输出到窗体上。

实现思想：

顺序表是用一组地址连续的存储单元依次存储线性表的数据元素。线性表是最常用且最简单的一种数据结构，一个线性表是 n 个数据元素的有限序列。

假设顺序表的每个元素需要占用 L 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则顺序表中第 i+1 个数据元素的存储位置 Loci+1 和第 i 个数据元素的存储位置 Loci 之间的关系如下：

```
Loci+1=Loci+L
```

顺序表的第 i 个数据元素的存储位置为：

```
Loci+1=Loc1+(i-1)*L
```

上面的式子中，Loc1 是顺序表的第一个元素的存储位置，通常称为顺序表的起始位置或是基地址。

例 13.9　声明 struct sqlist 类型，创建 InsertList()自定义函数，在顺序表中插入元素。实现步骤如下（光盘位置：光盘\MR\Instance\13\09）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
#define Listsize 100
```

（3）声明 struct sqlist 类型。

```
struct sqlist
{
    int data[Listsize];
    int length;
};
```

（4）创建 InsertList ()自定义函数，在顺序表中插入元素，代码如下：

```
void InsertList(struct sqlist *l, int t, int i)                    /*创建 Insert List()自定义函数*/
{
    int j;                                                         /*定义变量*/
    if (i < 0 || i > l->length)
    {
        printf("position error");
        exit(1);
    }
    if (l->length >= Listsize)                                     /*如果超出顺序表范围，则溢出*/
    {
        printf("overflow");
        exit(1);
    }
    for (j = l->length - 1; j >= i; j--)                           /*插入元素*/
        l->data[j + 1] = l->data[j];
    l->data[i] = t;
    l->length++;
}
```

（5）创建 main()函数作为程序的入口函数。

```
void main()
{
    struct sqlist *sq;
    int i, n, t;                                                   /*定义整型变量*/
    sq = (struct sqlist*)malloc(sizeof(struct sqlist));            /*分配空间*/
    sq->length = 0;
    printf("Please input the size of the list :");                 /*输出提示信息*/
    scanf("%d", &n);
    printf("Please input the elements of the list:\n");            /*输出提示信息*/
    for (i = 0; i < n; i++)
    {
        scanf("%d", &t);
        InsertList(sq, t, i);                                      /*插入元素*/
    }
    printf("Now the list is:\n");                                  /*输出提示信息*/
    for (i = 0; i < sq->length; i++)
    {
        printf("%d ", sq->data[i]);
    }
    getch();
}
```

运行本实例，效果如图 13.12 所示。

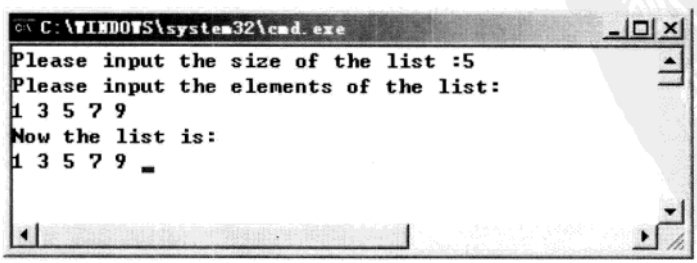


图 13.12　在数据库中保存大文本文件

技巧 10 向链表中插入节点

本实例实现创建一个指定节点数的链表，并在链表中插入一个新的节点。

实现思想：

在链表中插入节点需要解决两个问题：怎样找到插入的位置，怎样实现插入操作。

这里实现的是，在事先指定的位置插入一个指定数据元素的节点。因为链表是链式结构的，要想插入一个节点，就先要断开链表，才能将新节点插入。就像是一群小学生手拉手，要将一个新的小学生加入其中，就需要在要插入的位置先使两名小学生的手脱开，让前面的同学拉新同学的手，新同学拉后面同学的手。这样就完成了插入，形成一个新的队列。

按照上面的思路，如果在第 i 个位置前插入一个节点，就先要找到插入的位置，使用指针 p 指向该节点，本实例使用下面的循环语句从头节点移动指针，找到要插入位置的前一个节点。插入前的链表如图 13.13(a)所示。程序代码如下：

```
p = head;
while (p && j < i - 1)
{
    p = p->next;
    ++j;
}
```

将要插入节点（即 s 指向的节点）指向链表中插入位置的下一个节点。代码如下：

```
s->next = p->next;
```

将指针 p 指向的节点（即插入位置的前一个节点）指向要插入的节点，代码如下：

```
p->next = s;
```

插入节点后的链表如图 13.12（b）所示。

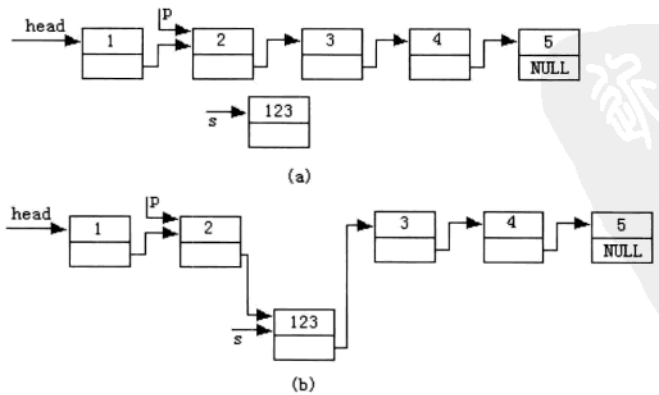


图 13.13 单链表插入节点示意图

例 13.10　声明 struct student 类型，创建 insertnode ()自定义函数，实现在已知链表中插入一个节点。创建 main()函数作为程序的入口函数，调用 create()和 insertnode()自定义函数实现创建链表并在链表中插入一个节点。实现步骤如下所示（光盘位置：光盘\MR\Instance\13\10）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）声明 struct student 类型。

```
struct student
{
    int num;
    struct student *next;
};
```

（4）创建 insertnode ()自定义函数，实现在已知链表中插入一个节点。程序代码如下：

```
struct student *insertnode(struct student *head, char x, int i)
{
    int j = 0;                                              /*定义整型变量*/
    struct student *p,*s;
    p = head;
    while (p && j < i - 1)                                  /*找到要插入的位置*/
    {
        p = p->next;
        ++j;                                                /*先对 j 进行++，然后再用 j 进行运算*/
    }
    if (!p || j > i - 1)
        exit(1);
    s = (struct student*)malloc(sizeof(struct student));    /*申请节点空间*/
    s->num = x;                                             /*为数据域赋值*/
    s->next = p->next;
    p->next = s;
}
```

（5）创建 main()函数作为程序的入口函数，调用 create()和 insertnode ()自定义函数实现创建链表并在链表中插入一个节点。程序代码如下：

```
void main()
{
    int n, i;                                               /*定义整型变量*/
    int x;
    struct student *q;                                      /*声明节点类型的指针变量*/
    printf("Input the count of the nodes you want to creat:");  /*输出提示信息*/
    scanf("%d", &n);
    q = create(n);                                          /*创建链表*/
    i = 2;
    x = 123;
    insertnode(q, x, i);                                    /*插入节点*/
    printf("The result is:\n");
```

```
    while (q)                                          /*输出插入节点后的链表*/
    {
        printf("%d  ", q->num);
        q = q->next;
    }
}
```

运行本实例，效果如图 13.14 所示。

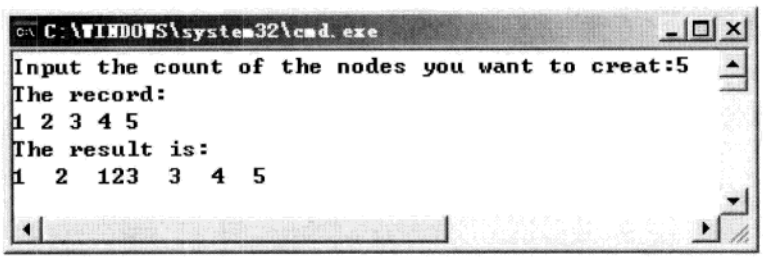


图 13.14　向链表中插入节点

技巧 11　从链表中删除节点

技巧 10 是关于如何插入节点，而本技巧要实现创建一个链表，并在链表中删除一个节点，然后将删除节点后的链表显示在窗体上。

实现思想：

在链表中删除节点就是将节点从链表中分离出来，撤销原来的链接关系，而不是将节点从内存中抹掉。就像是有编号为 1、2、3、4、5 的 5 个小学生手拉手，现在要将编号为 3 的小学生从这个队列中分离出来。只要将编号为 3 的小学生的手从两边脱开，使编号为 1 和编号为 2 的两个小学生拉手，形成一个新的队列。图 13.15（a）所示为原来队列，图 13.15（b）为分离一个学生后的队列。



图 13.15　从队列中删除元素

从链表中删除一个节点与上面相仿，只是需要改变要删除节点的前一个节点的指针域，使其指向要删除节点的下一个节点即可。其示意图如图 13.16 所示。

例 13.11　声明 struct student 类型，创建 create()自定义函数实现创建链表，创建 delnode()自定义函数，实现删除已知链表中的指定节点，并释放删除节点空间。创建 main()函数作为程

序的入口函数，实现删除链表中第 3 个节点。实现的具体步骤如下（光盘位置：光盘\MR\Instance\13\11）。

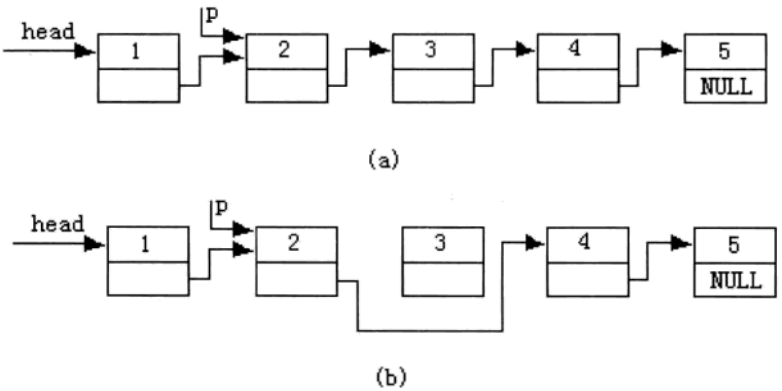


图 13.16　从链表中删除节点示意图

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）声明 struct student 类型。

```
typedef struct student
{
    int num;
    struct student *next;
} LNode,*LinkList;
```

（4）创建 create()自定义函数实现创建链表，代码如下：

```
struct student *create(int n)
{
    int i;                                                          /*定义整型变量*/
    struct student *head,*p1,*p2;
    int a;                                                          /*定义整型变量*/
    char b;                                                         /*定义字符型变量*/
    head = NULL;                                                    /*初始化头节点地址*/
    printf("The record:\n");
    for (i = n; i > 0; --i)
    {
        p1 = (struct student*)malloc(sizeof(struct student));       /*分配空间*/
        scanf("%d", &a);
        p1->num = a;                                                /*数据域赋值*/
        if (head == NULL)
        {
            head = p1;
```

```
            p2 = p1;
        }
        else
        {
            p2->next = p1;                                  /*指定后继指针*/
            p2 = p1;
        }
    }
    p2->next = NULL;                                        /*后继指针为空，链表结束*/
    return head;                                            /*返回头节点*/
}
```

（5）创建 delnode()自定义函数，实现删除已知链表中的指定节点，并释放删除节点空间。程序代码如下：

```
struct student *delnode(struct student *head, int i)
{
    int j = 0;                                              /*定义整型变量，并进行初始化*/
    struct student *p, *r;
    p = head;
    while (p && j < i - 1)                                  /*找到要删除的位置*/
    {
        p = p->next;
        ++j;
    }
    if (!p->next || j > i - 1)
        exit(1);
    r = p->next;
    p->next = r->next;                                      /*删除节点*/
    free(r);
}
```

（6）创建 main()函数作为程序的入口函数，调用 create()自定义函数实现创建链表，调用 delnode()自定义函数实现删除链表中第 3 个节点。程序代码如下：

```
void main()
{
    int n, i;                                               /*定义整型变量*/
    int x;                                                  /*定义整型变量*/
    struct student *q;                                      /*定义指针*/
    printf("Input the count of the nodes you want to creat:");  /*输出提示信息*/
    scanf("%d", &n);                                        /*输入要创建的链表节点数*/
    q = create(n);                                          /*创建链表*/
    i = 2;
    delnode(q, i);                                          /*删除第 3 个节点*/
    printf("The third record is deleted!\nThe result is:\n");  /*输出提示信息*/
    while (q)                                               /*输出删除节点后的链表*/
    {
        printf("%d  ", q->num);
        q = q->next;
    }getch();
}
```

运行本实例，效果如图 13.17 所示。

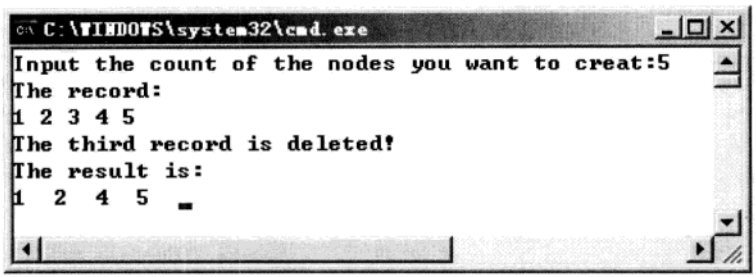


图 13.17　从链表中删除节点

技巧 12　合并两个链表

本实例实现将两个链表合并，合并后的链表为原来两个链表的连接，即将第二个链表直接连接到第一个链表的尾部，合成为一个链表。

实现思想：

本实例是将两个链表合并，即将两个链表连接起来。主要的思想是先找到第一个链表的尾节点，使其指针域指向下一个链表的头节点。

例 13.12　声明 struct student 类型，创建 create()自定义函数，实现创建一个链表，创建 coalition()自定义函数，实现将创建的两个链表合并。创建 main()函数作为程序的入口函数，实现将两个链表合并，并将合并后的链表输出。具体实现的步骤如下（光盘位置：光盘\MR\Instance\13\12）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）声明 struct student 类型。

```
typedef struct student
{
    int num;
    struct student *next;
} LNode,*LinkList;
```

（4）创建 create()自定义函数，实现创建一个链表，其返回值为指针值，返回值指向一个 struct node 类型数据，实际上是返回链表的头指针。

```
LinkList create(void)                                    /*自定义函数*/
{
    LinkList head;
    LNode *p1,*p2;                                       /*声明指针*/
    char a;                                              /*定义字符型变量*/
    head = NULL;
```

```
    a = getchar();
    while (a != '\n')
    {
        p1 = (LNode*)malloc(sizeof(LNode));                    /*分配空间*/
        p1->num = a;                                           /*数据域赋值*/
        if (head == NULL)
            head = p1;
        else
            p2->next = p1;
        p2 = p1;
        a = getchar();
    }
    p2->next = NULL;
    return head;
}
```

（6）创建 coalition()自定义函数，实现将创建的两个链表合并，程序代码如下：

```
LinkList coalition(LinkList L1, LinkList L2)                   /*创建自定义函数*/
{
    LNode *temp;                                               /*声明指针*/
    if (L1 == NULL)
        return L2;                                             /*返回变量 L2*/
    else
    {
        if (L2 != NULL)
        {
            for (temp = L1; temp->next != NULL; temp = temp->next);
            temp->next = L2;                                   /*遍历 L1 中节点直到尾节点*/
        }
    }
    return L1;
}
```

（7）创建 main()函数作为程序的入口函数，调用 create()自定义函数实现创建两个链表，调用 coalition ()自定义函数实现将两个链表合并，并将合并后的链表输出。程序代码如下：

```
void main()
{
    LinkList L1, L2, L3;
    printf("Please input two linklist:\n");                    /*输出提示信息*/
    printf("The first linklist:\n");                           /*输出提示信息*/
    L1 = create();                                             /*创建一个链表*/
    printf("The second linklist:\n");
    L2 = create();                                             /*创建第二个链表*/
    coalition(L1, L2);                                         /*连接两个链表*/
    printf("The resultant linklist is:\n");
    while (L1)                                                 /*输出合并后的链表*/
    {
        printf("%c", L1->num);
        L1 = L1->next;
    }
    getch();
}
```

运行本实例，效果如图 13.18 所示。

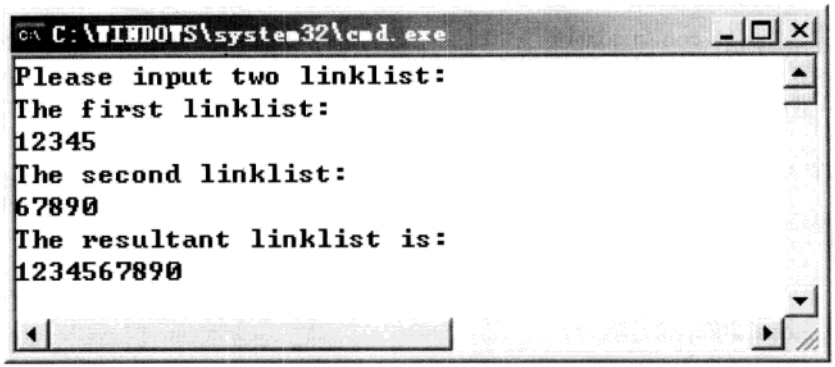


图 13.18　合并两个链表

技巧 13　应用栈实现进制转换

本技巧实现应用栈实现进制的转换，可以将十进制数转换为其他进制数。

实现思想：

本实例使用栈实现进制的转换。由于栈具有后进先出的固有特性，使栈成为了程序设计中有用的工具。这里实现的是十进制数 n 和其他进制数 d 的转换，其解决方法很多。

本技巧的算法思想为：将十进制数与要转换的进制数值做整除运算，取余，将整除运算得到的商再与要转换的进制数值做整除运算，再取余，重复以上操作，直到除尽，最后将得到的余数逆序排列，即是要得到的结果。

例如要将十进制数 1000 转换为八进制数，其运算过程如表：

N	N div 8	N mod 8
1000	125	0
125	15	5
15	1	7
1	0	1

其中 div 为整除运算，mod 为求余运算。

因为进制转换结果是上述取余运算的逆序序列，这正符合了栈的后进先出的特性。先将每次运算所得的余数一味地入栈，然后一味地出栈，得到的出栈序列即是所要的结果。

提示：栈是顺序表的特殊形式。

例 13.13 声明 struct student 类型。创建自定义函数实现构造栈、进栈、出栈、取栈顶元素等操作，创建自定义函数 conversion()实现十进制整数的进制转换功能。创建 main()函数作为程序的入口函数，调用 conversion ()自定义函数实现对给定的十进制整数进行进制转换（光盘位置：光盘\MR\Instance\13\13）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）声明 struct student 类型。

```
typedef struct{
      DataType *base;
      DataType *top;
      int stacksize;
}SeqStack;
```

（4）创建自定义函数实现构造栈、进栈、出栈、取栈顶元素等操作。实现代码如下：

```
void Initial(SeqStack *s)
{/*构造一个空栈*/
      s->base=(DataType *)malloc(STACK_SIZE * sizeof(DataType));
      if(!s->base) exit (-1);
      s->top=s->base;
      s->stacksize=STACK_SIZE;

}
/*判栈空*/
int IsEmpty(SeqStack *S)
{
    return S->top==S->base;
}
/*判栈满*/
int IsFull(SeqStack *S)
{
    return S->top-S->base==STACK_SIZE-1;
}
/*进栈*/
void Push(SeqStack *S,DataType x)
{
    if (IsFull(S))
    {
        printf("栈上溢"); /*上溢，退出运行*/
        exit(1);
    }
  *S->top++ =x;/*栈顶指针加 1 后将 x 入栈*/
}
/*出栈*/
DataType Pop(SeqStack *S)
{
    if(IsEmpty(S))
    {
        printf("栈为空"); /*下溢，退出运行*/
        exit(1);
    }
```

```
    return *--S->top;/*栈顶元素返回后将栈顶指针减 1*/
}
/* 取栈顶元素*/
DataType Top(SeqStack *S)
{
    if(IsEmpty(S))
    {
        printf("栈为空");                                    /*下溢，退出运行*/
        exit(1);
    }
    return *(S->top-1);
}
```

创建自定义函数 conversion()实现十进制整数的进制转换功能。实现代码如下：

```
void conversion (int N,int B)
{                                                    /*假设 N 是非负的十进制整数，输出等值的 B 进制数*/
    int i;
    SeqStack *S;
    Initial(S);
    while(N)
    {                                                /*从右向左产生 B 进制的各位数字，并将其进栈*/
        Push(S,N%B); /*将 bi 进栈 0<=i<=j*/
        N=N/B;
    }
    while(!IsEmpty(S))
    {                                                /*栈非空时退栈输出*/
        i=Pop(S);
        printf("%d",i);
    }
}
```

（5）创建 main()函数作为程序的入口函数，调用 conversion ()自定义函数实现对给定的十进制整数进行进制转换。程序代码如下：

```
void main()
{
    int n,d;                                         /*定义整型变量*/
    printf("Input the integer you want to transform:");  /*输出提示信息*/
    scanf("%d",&n);
    printf("Input the integer of the system: ");     /*输出提示信息*/
    scanf("%d",&d);
    printf("result:");                               /*输出提示信息*/
    conversion(n,d);
    getch();
}
```

运行本实例，效果如图 13.19 所示。

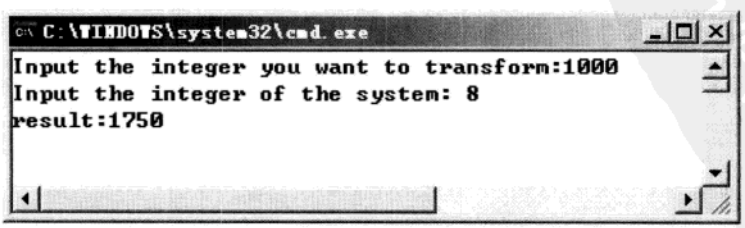


图 13.19　应用栈实现进制转换

技巧 14 括号匹配检测

在实际应用选择判断语句和循环语句时，常常在括号匹配处出错，本技巧要求编写检测括号是否匹配的程序，它的主要功能是对输入的一组字符串进行检测，当输入的字符串中括号（包括“{}”、“[]”、“()”）匹配时输出 matching，否则输出 No matching。

实现思想：

[(36-21)*56-23/(23+12)]，观察该字符串会发现如果从右向左扫描，那么每个右括号（包括“}”、“]”及“)”）将与最近的那个未配对的左括号相配对。从这个分析中我们可以想到一种方法，就是对输入的每个字符进行判断。

如果输入的是左括号中的任意一种就进行进栈操作，如输入的是右括号中的任意一种，则进行取栈顶元素操作，看取出的元素是否是与这个右括号相匹配的元素，如果是则令其标志作用的变量置 1，否则置 0。

若标志变量有一次为 0 就不需要进行下面的操作说明该字符串不匹配。当对整个字符串中的字符均进行了判断后看标志位是否为 1 且该栈是否为空，若标志位为 1 且该栈为空，则说明该字符串括号匹配，否则不匹配。

例 13.14 创建一个类，在该类中编写一个方法，用于获得数据库中的所有表名称 (光盘位置：光盘\MR\Instance\13\14)。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件、进行宏定义及数据类型的指定。

```
#include <stdio.h>
#define STACK_SIZE 100                          /*假定预分配的栈空间最多为 100 个元素*/
typedef char DataType;                          /*设定 DataType 的代表的数据类型为字符型*/
```

（3）定义结构体类型及变量。

```
typedef struct                                  /*定义结构体*/
{
    DataType *base;                             /*定义栈底指针*/
    DataType *top;                              /*定义栈顶指针*/
    int stacksize;                              /*定义栈的大小*/
} SeqStack;                                     /*SeqStack 为该结构体类型*/
```

（4）创建自定义函数实现构造栈、进栈、出栈、取栈顶元素等操作。实现代码如下：

```
void Initial(SeqStack *S) /*初始化栈*/
{
    S->base = (DataType*)malloc(STACK_SIZE *sizeof(DataType));
    if (!S->base)
        exit( - 1);
```

```
    S->top = S->base;                                   /*栈为空时栈顶栈底指针指向同一处*/
    S->stacksize = STACK_SIZE;
}
int IsEmpty(SeqStack *S)                                /*判栈空*/
{
    return S->top == S->base;
}
int IsFull(SeqStack *S)                                 /*判栈满*/
{
    return S->top - S->base == STACK_SIZE - 1;
}

void Push(SeqStack *S, DataType x)                      /*进栈*/
{
    if (IsFull(S))
    {
        printf("overflow");                             /*上溢，退出运行*/
        exit(1);
    }
    else
        *S->top++ = x;                                  /*栈顶指针加 1 后将 x 入栈*/
}
void Pop(SeqStack *S)                                   /*出栈*/
{
    if (IsEmpty(S))
    {
        printf("NULL");                                 /*下溢，退出运行*/
        exit(1);
    }
    else
        --S->top;                                       /*栈顶元素返回后将栈顶指针减 1*/
}
DataType Top(SeqStack *S)                               /* 取栈顶元素*/
{
    if (IsEmpty(S))
    {
        printf("empty");                                /*下溢，退出运行*/
        exit(1);
    }
    return *(S->top - 1);
}
```

（5）创建自定义函数 match()实现行编辑功能。实现代码如下：

```
int match(SeqStack *S, char *str)
{
    char x;                                             /*定义字符型变量*/
    int i, flag = 1;                                    /*定义整型变量，并对变量 flag 进行初始化*/
    for (i = 0; str[i] != '\0'; i++)                    /*只要 str[i]不是'\0'时执行循环体*/
    {
        switch (str[i])
        {
            case '(':
                Push(S, '(');
                break;
            case '[':
                Push(S, '[');
                break;
```

```
            case '{':
                Push(S, '{');
                break;
            case ')':
                x = Top(S);
                Pop(S);
                if (x != '(')
                    flag = 0;
                break;
            case ']':
                x = Top(S);
                Pop(S);
                if (x != '[')
                    flag = 0;
                break;
            case '}':
                x = Top(S);
                Pop(S);
                if (x != '{')
                    flag = 0;
                break;
        }
        if (!flag)
            break;
    }
    if (IsEmpty(S) == 1 && flag)
        return 1;
    else
        return 0;
}
```

（6）主函数程序代码如下：

```
main()                                          /*main()主函数*/
{
    SeqStack *st;                               /*声明指针*/
    char str[100];                              /*定义字符型数组*/
    Initial(st);
    gets(str);
    if (match(st, str))
        printf("matching\n");
    else
        printf("no matching\n");
}
```

运行本实例，效果如图 13.20 所示。

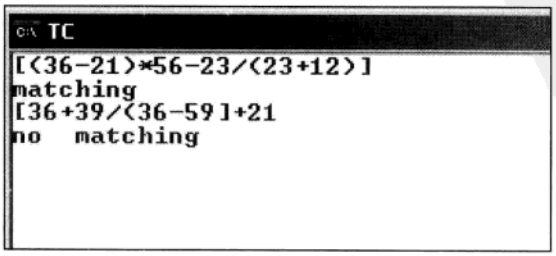


图 13.20　括号匹配检测

技巧 15　串的模式匹配

分别从键盘中输入主串和模式串，如果能够找到子串在主串的位置，则将该位置输出，否则匹配不成功，输出提示信息。

实现思想：

算法的基本思想是：分别设置计数指针 i 和 j 指示主串和模式串当前正待比较的字符位置，从主串的第一个字符和模式的第一个字符开始比较，若相等，则继续逐个比较后续字符；否则从主串的下一个字符起再重新和模式串的字符比较。

依次类推，直到找到主串中的一串连续字符和模式串中的字符相等，则说明模式匹配成功，否则匹配不成功。

例 13.15　定义结构体用来存储串的相关信息，自定义 firststr()函数来实现模式匹配检测。在 main()主函数中调用自定义函数，实现串的模式匹配，并将结果输出。实现步骤如下所示（光盘位置：光盘\MR\Instance\13\15）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件并进行宏定义。

```
#include <string.h>
#include <stdio.h>
#define Maxsize 100
```

（3）定义结构体用来存储串的相关信息。代码如下：

```
typedef struct                                  /*定义结构体，用来存储串的相关信息*/
{
    char string[Maxsize];
    int len;
} str;                                          /*定义 str 为该结构体类型*/
```

（4）自定义 firststr()函数来实现模式匹配检测。代码如下：

```
int findstr(str s, str t)                       /*自定义函数 findstr*/
{
    int i = 1, j = 0, k = 0, pos;
    while (i < s.len && j < t.len)
    {
        if (s.string[k] == t.string[j])         /*判断主串与模式串对应元素是否匹配*/
        {
            k++;                                /*主串向后移一位*/
            j++;                                /*模式串向后移一位*/
        }
        else
        {   /*主串和模式串重新退回，从主串的下一个位置开始下一次匹配*/
            i++;
            k = i;
            j = 0;
```

```
        }
    }
    if (j == t.len)                                    /*判断模式串是否已经匹配到最后一个字符*/
        pos = k - j + 1;                               /*指向匹配成功的第一个字符*/
    else
        pos =  - 1;
    return (pos);
}
```

（5）主函数程序代码如下：

```
main()
{
    str s, t;
    int p;
    printf("qing shu ru zhu cuan:\n");
    scanf("%s", s.string);                             /*从键盘中输入主串*/
    s.len = strlen(s.string);
    printf("qing shu ru mo shi cuan:\n");
    scanf("%s", t.string);                             /*从键盘中输入模式串*/
    t.len = strlen(t.string);                          /*获取模式串的长度*/
    p = findstr(s, t);                                 /*调用 findstr 函数返回值赋给 p*/
    if (p ==  - 1)
        printf("no matching!");                        /*返回值如果为-1 说明未匹配*/
    else
        printf("matching!the position is: %d", p);     /*将匹配位置输出*/
}
```

运行本实例，效果如图 13.21 所示。

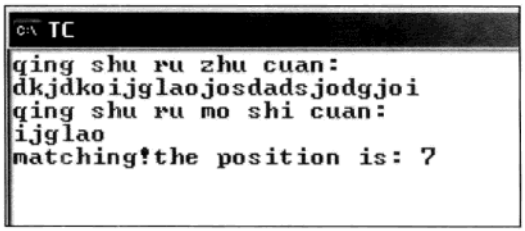


图 13.21　串的模式匹配

技巧 16　简单的文本编辑器

本程序要求实现 3 个功能：第一，要求对指定行输入字符串；第二，删除指定行的字符串；第三，显示输入的字符串的内容。

实现思想：

串是由零个或多个字符组成的有限序列，对串的存储可以有两种方式：一种是静态存储，另一种是动态存储。其中动态存储结构有两种方式：一种是链式存储结构，另一种是堆结构存储方式；这里着重说一下链式存储结构。

串的链式存储结构是包含数据域和指针域的节点结构。因为每个节点仅存放一个字符，这样比

较浪费空间，为了节省空间，可使每个节点存放若干个字符，这种结构叫作块链结构，本实例就是采用这种块链结构来实现的。

例 13.16　定义结构体和 5 个自定函数，分别实现每行头节点的初始化、选择菜单、向指定行中输入字符串、对指定行的删除、将输入的内容显示在屏幕上。在 main()主函数中调用自定义函数，实现简单的文本编译器功能。实现步骤如下所示（光盘位置：光盘\MR\Instance\13\16）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件并进行宏定义并对自定义函数进行声明。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
void Init();
void input();
void Delline();
void List();
int Menu();
```

（3）定义结构体用来存储每行字符串的相关信息并声明结构体类型数组 Head。代码如下：

```
typedef struct node                                    /*定义存放字符串的节点*/
{
    char data[50];
    struct node *next;
} strnode;
typedef struct head                                    /*定义每行的头节点*/
{
    int number;                                        /*行号*/
    int length;                                        /*字符串的长度*/
    strnode *next;
} headnode;
headnode Head[MAX];                                    /*定义有 100 行*/
```

（4）自定义 init()函数来实现每行头节点的初始化。代码如下：

```
void Init()                                            /*定义初始化函数*/
{
    int i;                                             /*定义整型变量*/
    for (i = 0; i < MAX; i++)
    {
        Head[i].length = 0;
    }
}
```

（5）自定义 Menu()函数来实现选择菜单，并将选择的菜单所对应的序号返回。代码如下：

```
int Menu()                                             /*定义菜单*/
{
    int i;
    i = 0;
    printf(" 1. Input\n");                             /*输出提示信息*/
    printf(" 2. Delete\n");                            /*输出提示信息*/
    printf(" 3. List\n");                              /*输出提示信息*/
    printf(" 4. Exit\n");                              /*输出提示信息*/
    while (i <= 0 || i > 4)
```

```
    {
        printf("please choose\n");
        scanf("%d", &i);
    }
    return i;
}
```

（6）自定义 input()函数来实现向指定行中输入字符串。代码如下：

```
void input()                                                    /*自定义输入字符串函数*/
{
    strnode *p,*find();                                         /*声明指针*/
    int i, j, LineNum;                                          /*定义整型变量*/
    char ch;                                                    /*定义字符型变量*/
    while (1)
    {
        j =-1;
        printf("input the number of line(0~100),101-exit:\n");
        scanf("%d", &LineNum);                                  /*输入要输入的字符串所在的行号*/
        if (LineNum < 0 || LineNum >= MAX)
            return ;
        printf("please input,#-end\n");
        i = LineNum;
        Head[i].number = LineNum;
        Head[i].next = (strnode*)malloc(sizeof(strnode));       /*分配内存空间*/
        p = Head[i].next;
        ch = getchar();
        while (ch != '#')
        {
            j++;                                                /*计数*/
            if (j >= 50)
             /*如果字符串长度超过 50 需要再分配一个节点空间*/
            {
                p->next = (strnode*)malloc(sizeof(strnode));
                p = p->next;                                    /*p 指向新分配的节点*/
            }
            p->data[j % 50] = ch; /*将输入的字符串放入 data 中*/
            ch = getchar();
        }
        Head[i].length = j + 1; /*长度*/
    }
}
```

（7）自定义 Delline()函数来实现对指定行的删除。代码如下：

```
void Delline()                                                  /*自定义删除行函数*/
{
    strnode *p,*q;                                              /*声明指针*/
    int i, LineNum;                                             /*定义整型变量*/
    while (1)
    {
        printf(
            "input the number of line which do you want to delete(0~100),101-exit:\n");
        scanf("%d", &LineNum);                                  /*输入要删除的行号*/
        if (LineNum < 0 || LineNum >= MAX)
                                                                /*如果超出行的范围则返回菜单界面*/
            return ;
        i = LineNum;
        p = Head[i].next;
        if (Head[i].length > 0)
        while (p != NULL)
```

```
        {
            q = p->next;
            free(p);                                    /*将 p 的空间释放*/
            p = q;
        }
        Head[i].length = 0;
        Head[i].number = 0;
    }
}
```

（8）自定义 list()函数来实现将输入的内容显示在屏幕上，具体代码如下：

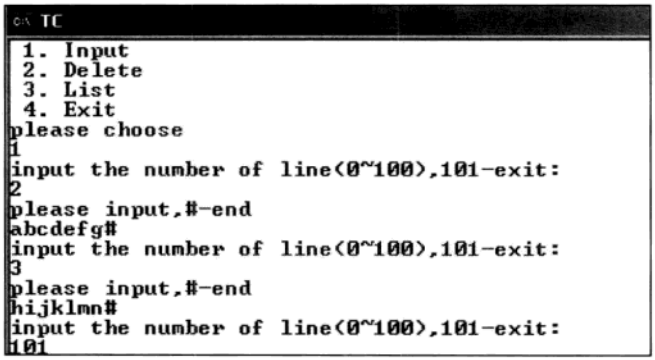
```
void List()
{
    strnode *p;                                         /*声明指针*/
    int i, j, m, n;                                     /*定义整型变量*/
    for (i = 0; i < MAX; i++)
    {
        if (Head[i].length > 0)
        {
            printf("line %d", Head[i].number);
            n = Head[i].length;
            m = 1;
            p = Head[i].next;
            for (j = 0; j < n; j++)
                if (j >= 50 *m)
            /*以 50 为准，超过一个则指向下一个节点*/
            {
                p = p->next;
                m++;                                    /*节点个数*/
            }
            else
                printf("%c", p->data[j % 50]);
                                                        /*将节点中内容输出*/
            printf("\n");
        }
    }
    printf("\n");                                       /*换行*/
}
```

（9）主函数程序代码如下：

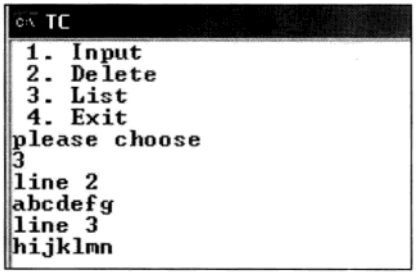
```
main()
{
    int sel;                                            /*定义整型变量*/
    Init();                                             /*初始化*/
    while (1)
    {
        sel = Menu();
        switch (sel)                                    /*输入对应数字进行选择*/
        {
            case 1:
                input();
                break;
            case 2:
                Delline();
                break;
            case 3:
                List();
                break;
            case 4:
```

```
                exit(0);
            }
        }
    }
```

运行本实例，效果如图 13.22 所示。



（a）向指定行输入字符串界面



（b）显示输入的字符串内容

图 13.22　简单的文本编辑器



第 14 章　C 语言开发人员必知必会的高级开发技巧

在进行 C 语言程序开发时，常常需要用到大量的库函数，而且 C 语言可以实现的技巧都与系统相关，因此本章根据笔者的经验，对文件和网络开发中的一些技巧进行了讲解。

这些技巧包括设置 DOS 系统时间和日期、函数实现字符匹配、查找位置信息、建立目录、拷贝目录、删除目录、访问系统 temp 中的文件、设置组合键、获取当前日期与时间、设置系统日期、备份/恢复硬盘分区表等，通过学习本章，可以帮助 C 语言的开发人员在进行库函数和系统相关的开发时提高开发效率。

技巧 01　设置 DOS 系统时间

编程实现将系统当前时间改为 10：5：12，要求将原来时间和修改后的时间均显示在屏幕上。

实现思想：

本程序中用到了 settime()函数，具体使用说明如下：

settime()函数的一般形式为：

```
void settime (struct time *t)
```

该函数的作用是按照 t 指向的结构中指定的值设置 dos 系统时间。该函数的原型在 dos.h 中。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）程序中分别调用 gettime 函数和 settime 函数分别来获取系统当前时间及按照指定的值重新设置系统时间。

例 14.1　在主函数中调用 gettime 函数和 settime 函数分别来获取系统当前时间及按照指定的

值重新设置系统时间(光盘位置：光盘\MR\Instance\14\01)。

```
#include <dos.h>
#include <stdio.h>
main()
{
    struct time sett, now;                                          /*定义 sett、now 为 time 结构体变量
*/
    struct time origint;                                            /*定义 origint 为 time 结构体变量*/
    gettime(&origint);                                              /*获取系统当前时间*/
    /*输出系统当前时间*/
    printf("original time is:%d:%d:%d\n", origint.ti_hour, origint.ti_min,origint.ti_sec);
    sett.ti_hour = 10;                                              /*设置系统时间中小时为 10*/
    sett.ti_min = 5;                                                /*设置系统时间中分为 5*/
    sett.ti_sec = 12;                                               /*设置系统时间中秒为 12*/
    sett.ti_hund = 32;
    /*使用 settime()按照指定的数据对系统时间进行设置*/
    settime(&sett);
    gettime(&now);
    /*输出设置后的系统时间*/
    printf("time after setting is:%d:%d:%d", now.ti_hour, now.ti_min,now.ti_sec);
    getch();
}
```

运行本实例，效果如图 14.1 所示。

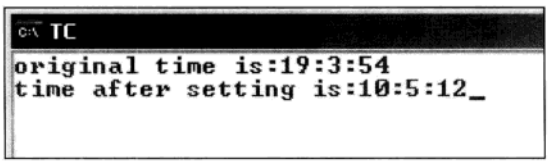


图 14.1　设置 DOS 系统时间

技巧 02　设置 DOS 系统日期

编程实现将系统当前日期改为 2008 年 10 月 23 日，要求将原来日期和修改后的日期均显示在屏幕上。

实现思路：

本程序中用到了 setdate()函数，具体使用说明如下：

setdate()函数一般形式如下：

```
void setdate(struct date *d)
```

该函数的作用是按照 d 指向的结构中指定的值设置 dos 系统日期。该函数的原型在 dos.h 中。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）程序中分别调用 getdate 函数和 setdate 函数分别来获取系统当前日期及按照指定的值重新设置系统日期。

例 14.2　调用 getdate 函数和 setdate 函数分别来获取系统当前日期及按照指定的值重新设置系统日期(光盘位置：光盘\MR\Instance\14\02)。

```
#include <dos.h>
#include <stdio.h>
main()
{
    struct date setd,now;                                          /*定义 setd 为 date 结构体变量*/
    struct date origind;                                           /*定义 origind 为 date 结构体变量*/
    getdate(&origind);                                             /*获取系统当前日期*/
    /*输出系统当前日期*/
    printf("original data is:%d--%d--%d\n", origind.da_year, origind.da_mon,origind.da_day);
    setd.da_year = 2008;                                           /*设置系统日期中年份为 2008*/
    setd.da_mon = 10;                                              /*设置系统日期中月份为 10*/
    setd.da_day = 23;                                              /*设置系统日期中日为 23*/
    /*使用 setdate()按照上面指定的数据对系统时间进行设置*/
    setdate(&setd);
    getdate(&now)                                                  /*获取系统重新设置后的当前日期*/
    /*输出设置后的系统时间*/
    printf("date after setting is:%d--%d--%d", now.da_year, now.da_mon,now.da_day);
    getch();
}
```

运行本实例，效果如图 14.2 所示。

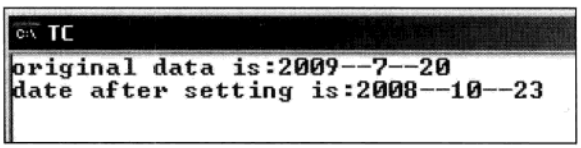


图 14.2　复制文件窗体

技巧 03　函数实现字符匹配

从键盘中输入一字符串，再输入要查找的字符，在屏幕上输出字符在字符串中的位置并将从该字符开始的余下字符串内容全部输出。

实现思路：

本程序中用到了 strchr()函数，具体使用说明如下：

strchr()函数一般形式如下：

```
char *strchr(char *str,char ch)
```

该函数的作用是返回由 str 所指向的字符串中首先出现 ch 的位置指针，如果未发现与 ch 匹配

的字符，则返回空指针。函数的原型在 string.h 中。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）从键盘中输入一字符串，再输入要查找的字符，调用 strchr()函数，返回指向该字符在字符串中首次出现的位置。

例 14.3 应用 strchr()函数实现字符匹配，具体代码如下所示（光盘位置：光盘\MR\Instance\14\03）。

```
#include <stdio.h>
#include <string.h>
main()
{
    char string[50];
    char *str, ch;
    printf("please input string:\n");
    gets(string);                                                    /*输入字符串到数组 str 中*/
    printf("please input the character which do you want to find:\n");
    scanf("%c", &ch);                                                /*输入要进行匹配的字符*/
    str = strchr(string, ch);                                        /*调用 strchr()函数*/
    if (str)                                                         /*判断返回的指针是否为空*/
    {
        printf("the character %c is at position:%d\n", ch, str - string);  /*将字符出现的位置输出*/
        printf("the rest string from %c is:%s\n", ch, str);          /*输出余下字符*/
    }
    else
        printf("the character was not found.\n");                    /*当未找到时输出提示信息*/
}
```

运行本实例，效果如图 14.3 所示。

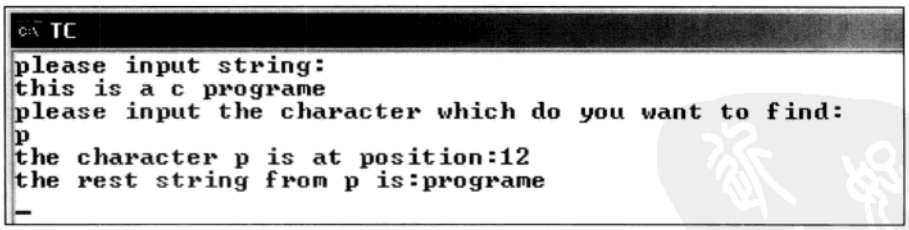


图 14.3 函数实现字符匹配

技巧 04 查找位置信息

从键盘中输入 str1 和 str2，查找 str1 字符串中第一个不属于 str2 字符串中字符的位置，并将该位置输出；再从键盘中输入 str3 和 str4，查找在 str3 中是否包含 str4，无论包含与否给出提示信息。

实现思想：

本程序中用到了 strspn()与 strstr()函数，具体使用说明如下：

strspn()函数一般形式如下：

```
char *strspn(char *str1, char *str2)
```

该函数的作用是在 str1 字符串中寻找第一个不属于 str2 字符串中字符的位置。该函数返回 str1 中第一个与 str2 任一个字符不相匹配的字符下标。该函数的原型在 string.h 中。

strstr()函数一般形式如下：

```
char *strstr(char *str1, char *str2)
```

该函数的作用是在字符串 str1 中寻找 str2 字符串的位置，并返回指向该位置的指针，如果没有找到相匹配的就返回空指针。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）从键盘中输入字符串 1 和字符串 3，调用 strspn()函数实现在 str1 字符串中寻找第一个不属于 str2 字符串中字符的位置，从键盘中输入字符串 3 和字符串 4，调用 strstr()函数实现在字符串 str3 中寻找 str4 字符串的位置，并返回指向该位置的指针。

例 14.4　调用 strspn()与 strstr()函数，实现查找位置信息，具体代码如下（光盘位置：光盘\MR\Instance\14\04）。

```
#include <string.h>
#include <stdio.h>
main()
{
    char str1[30], str2[30], str3[30], str4[30],*p;
    int pos;
    printf("please input string1:");
    gets(str1);                                          /*从键盘中输入字符串 1*/
    printf("please input string2:");
    gets(str2);                                          /*从键盘中输入字符串 2*/
    pos = strspn(str1, str2);                            /*调用函数 strspn 找出不同的位置*/
    printf("the position you want to find is:%d\n",pos);
    printf("please input string3:");
    gets(str3);                                          /*从键盘中输入字符串 3*/
    printf("please input string4:");
    gets(str4);                                          /*从键盘中输入字符串 4*/
    p = strstr(str3, str4);                              /*调用函数 strstr 查看 str3 中是否包含 str4*/
    if (p)
        printf("str3 include str4\n");
    else
        printf("can not find str4 in str3!");
}
```

运行本实例，效果如图 14.4 所示。

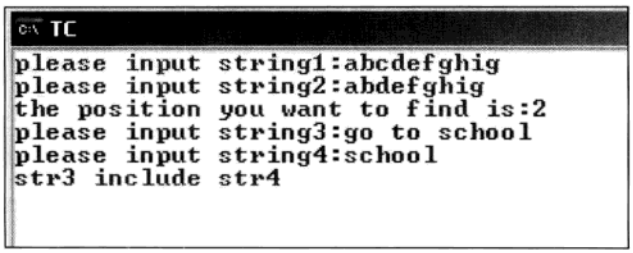


图 14.4 查找位置信息

技巧 05 建立目录

编程实现在当前目录下在创建一个目录，创建的文件夹名称为 temp。

实现思想：

本程序用到 mkdir()函数，具体使用说明如下：

mkdir()函数一般形式如下：

```
int mkdir(char *path)
```

该函数的作用是由 path 所指向的路径名建立一个目录。该函数如果成功，则返回 0，否则返回 -1。函数的原型在 dir.h 中。

例 14.5 创建一个类，在该类中创建一个能够复制整个文件夹的方法，然后进行测试（光盘位置：光盘\MR\Instance\14\05）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <dir.h>
#include <stdio.h>
```

（3）调用 mkdir()函数以 temp 为路径名建立一个目录。

（4）主函数程序代码如下：

```
main()
{
    if (!mkdir("temp"))                          /*调用函数应 temp 为路径名建立一个目录*/
        printf("directory test is created\n");   /*目录建成输出提示信息*/
    else
    {
        printf("unable to create new directory\n");   /*目录未建成也同样输出提示信息*/
        exit(1);
```

```
    }
    getch();
}
```

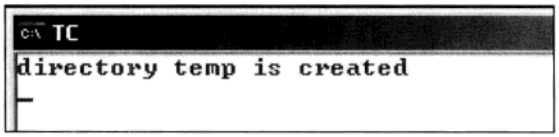
运行本实例，效果如图 14.5 所示。



（a）未创建目录前的目录



（b）创建 temp 目录后的目录



（c）程序运行界面

图 14.5　建立目录

技巧 06　复制当前目录

将当前的工作目录复制到数组 cdir 中并在屏幕上输出。

实现思想：

本程序用到 getdisk()函数和 getcurdir()函数，具体使用说明如下。

- getdisk()函数的一般形式为：

```
int getdisk(void)
```

该函数的作用是返回当前驱动器名的代码。驱动器 A 为 0，驱动器 B 为 1，依此类推。函数的原型在 dir.h 中。

- getcurdir()函数的一般形式为：

```
int getcurdir(int driver,char *dir)
```

该函数的作用是把有 driver 所指定的当前工作目录复制到由 dir 所指向的字符串中，若 drive 为零值，则指的是默认驱动器，驱动器 A 用 1，驱动器 B 用 2，依此类推。

由 dir 所指向的字符串长度必须至少为 MAXDIR 个字节。MAXDIR 为 dir.h 中定义的宏。目录名将不包含驱动器标识符，也不以反斜杠开始。函数的原型在 dir.h 中。

例 14.6　调用函数 getdisk()和 getcurdir()函数实现复制当前目录，具体步骤如下（光盘位置：

光盘\MR\Instance\14\06）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
#include <dos.h>
#include <dir.h>
```

（3）调用函数 getdisk()和 getcurdir()函数实现复制当前目录。

（4）主函数程序代码如下：

```
main()
{
    char cdir[MAXDIR];
    strcpy(cdir, "c:\\");
    cdir[0] = 'A' + getdisk();                          /*调用函数返回当前驱动器*/
    if (getcurdir(0, cdir + 3))
    /*调用函数将当前目录拷贝到 cdir+3 开始的数组中*/
    {
        printf("error");
        exit(1);
    }
    printf("the current directory is:%s\n", cdir);      /*输出提示信息*/
    getch();
}
```

运行本实例，效果如图 14.6 所示。

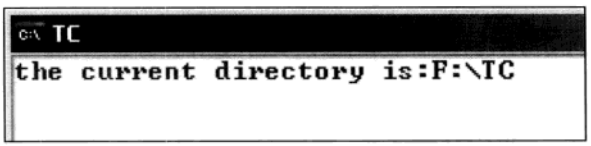


图 14.6　拷贝当前目录

技巧 07　删除目录

编程实现在当前目录下在创建一个目录，创建完成后按任意键将该目录删除。

实现思想：

本程序用到 mkdir()函数，具体使用说明如下。

rmdir()函数的一般形式为：

```
int rmdir(char *path)
```

该函数的作用是删除由 path 所指向的路径名和目录。删除时，目录必须是空的，必须不是当

前工作目录，也不能是根目录。函数的原型在 dir.h 中。

例 14.7　调用 mkdir()函数以 temp 为路径名建立一个目录，实现步骤如下（光盘位置：光盘\MR\Instance\14\07）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <dir.h>
#include <stdio.h>
```

（3）调用 mkdir()函数以 temp 为路径名建立一个目录。

（4）主函数程序代码如下：

```
main()
{
    char *name[10];                                          /*定义字符型数组存储文件名*/
    printf("please input directory name:\n");
    scanf("%s", name);                                       /*输入文件名*/
    printf("Press any key,and the directory will be removed\n");
    getch();
    if (!rmdir(name))                                        /*删除目录*/
        printf("Directory is removed.\n");                   /*删除成功输出提示信息*/
    else
        printf("can not remove");                            /*删除不成功输出提示信息*/
    getch();
}
```

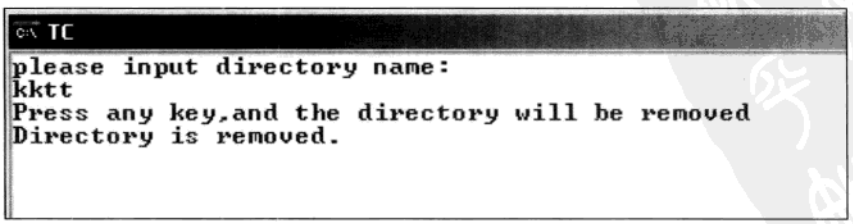
运行本实例，效果如图 14.7 所示。



（a）未删除目录前的目录



（b）删除 kktt 目录后的目录



（c）程序运行界面

图 14.7　删除目录

技巧 08 访问系统 temp 中文件

编程实现访问系统 temp 目录中的文件，并将文件内容显示在屏幕上。

实现思想：

本实例访问系统 temp 目录时使用了 getnev()函数，具体使用说明如下。

getnev()函数的一般形式为：

```
char *getnev(char *name)
```

该函数的作用是返回一个指向环境信息的指针，该信息与由 name 所指向的在 DOS 环境信息表中有关的字符串有关。如果用一个与任何环境数据都不匹配的参数调用 getnev()，则返回的是空指针。该函数的原型在 stdlib.h 中。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）调用 getnev()函数获取 temp 目录，再打文件夹中指定文件，利用 while 循环将文件中的内容输出到屏幕上。

例 14.8 调用 getnev()函数获取 temp 目录，实现访问系统 temp 中文件，具体代码如下（光盘位置：光盘\MR\Instance\14\08）。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    char *pathname, filename[30], ch;
    FILE *fp;
    pathname = getenv("TEMP");                          /*获取临时文件夹路径*/
    printf("\nCurrent path is:%s\n", pathname);          /*将临时文件夹路径输出*/
    printf("press any key to open one file!");
    getch();
    strcat(pathname, "\\new.txt");                      /*连接文件名*/
    strcpy(filename, pathname);                         /*将完整的文件路径名复制到 filename 中*/
    if ((fp = fopen(filename, "r")) != NULL)
    {
        printf("\nThe name of the file is");
        printf("\n%s", filename);                       /*输出文件路径名*/
        printf("\nThe content of the file is");
        printf("\n");
        ch = fgetc(fp);
        while (ch != EOF)                               /*读取文件中的内容*/
        {
            printf("%c", ch);
            ch = fgetc(fp);
```

```
        }
        fclose(fp);                                    /*关闭文件*/
    }
    else
        printf("can not open!");                       /*若文件未打开输出提示信息*/
}
```

运行本实例，效果如图 14.8 所示。

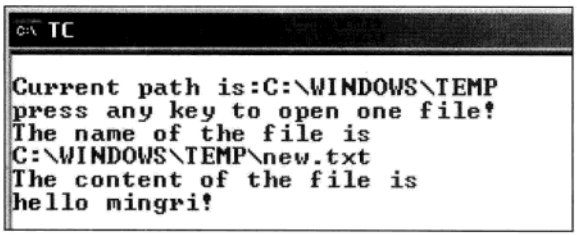


图 14.8　访问系统 temp 中文件

技巧 09　设置组合键

编程实现检测 Ctrl+Break 的当前状态，并设置 Ctrl+Break 状态显示在屏幕上。

实现思想：

本实例中分别使用了 getcbrk()函数和 setcbrk()函数来获取 Ctrl+Break 当前状态和设置当前 Ctrl+Break 状态，具体使用说明如下。

- getcbrk()函数的一般形式为：

```
int getcbrk(void)
```

该函数的作用是检测 Ctrl+Break 是关闭还是打开，当关闭时函数返回 0，当打开时函数返回 1。该函数的原型在 dos.h 中。

- setcbrk()函数的一般形式为：

```
int setcbrk(int cbrkvalue)
```

该函数的作用是设置 Ctrl+Break 的检测状态为 on 或 off。若 cbrkvalue=0，则检测状态置为 off，若 cbrkvalue=1，则检测状态置为 on。该函数的原型在 dos.h 中。

实现步骤：

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

（3）调用 getcbrk()函数和 setcbrk()函数来获取和设置当前 Ctrl+Break 状态，并将获取的状态输

出到屏幕上。

例 14.9 调用 getcbrk()函数和 setcbrk()函数，实现设置组合键，实现代码如下（光盘位置：光盘\MR\Instance\14\09）。

```
#include <stdio.h>
#include <dos.h>
main()
{
    /*检测当前 Ctrl+Break 状态并输出*/
    printf("\n The status of Ctrl+Break %s", (getcbrk()) ? "On" : "Off");
    printf("\npress any key to see the status of Ctrl+Break:");
    getch();
    setcbrk(1);                                          /*设置 Ctrl+Break 状态为 on*/
    /*检测当前 Ctrl+Break 状态并输出*/
    printf("\n The status of Ctrl+Break %s", (getcbrk()) ? "On" : "Off");
    printf("\npress any key to see the status of Ctrl+Break:");
    getch();
    setcbrk(0);                                          /*设置 Ctrl+Break 状态为 off*/
    /*检测当前 Ctrl+Break 状态并输出*/
    printf("\n The status of Ctrl+Break %s\n", (getcbrk()) ? "On" : "Off");
}
```

运行本实例，效果如图 14.9 所示。

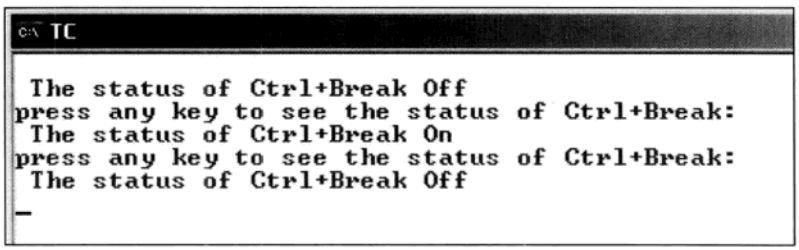


图 14.9 设置组合键

技巧 10 获取当前日期与时间

运行本实例编译后的可执行文件将会显示当地日期与时间。

实现思想：

要获取当地日期与时间，可以通过使用函数 localtime()来实现。该函数在 time.h 头文件中，其语法如下：

```
struct tm *localtime(const time_t *timer);
```

功能：把 timer 所指的时间（如函数 time 返回的时间）转换成当地标准时间，并以 tm 结构形式返回。

其中是要获取当前时间的传递参数，格式为 time_t 指针类型。

例 14.10　通过使用 localtime()实现获取当地日期与时间，实现步骤如下（光盘位置：光盘\MR\Instance\14\10）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdlib.h、time.h。

```
#include<stdio.h>
#include<time.h>
```

（3）主要程序代码。

```
main()
{
    struct tm * tmpointer;                                    // tm 结构指针
    time_t secs;                                              //声明 time_t 类型变量
    time(&secs);                                              //获取系统日期与时间
    tmpointer=localtime(&secs);                               //获取当地日期时间
    //输出本地时间
    printf("\nNow: %d-%d-%d %d:%d:%d",tmpointer->tm_mon,tmpointer->
    tm_mday,tmpointer->tm_year+1900,tmpointer->tm_hour,tmpointer->
    tm_min,tmpointer->tm_sec);
}
```

运行本实例，效果如图 14.10 所示。

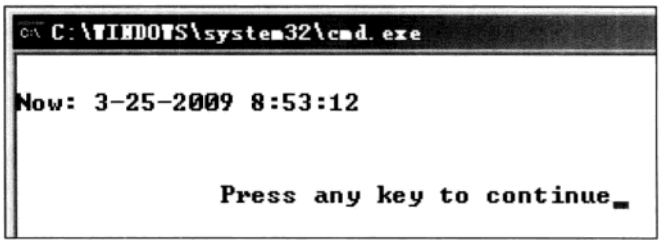


图 14.10　获取当前日期与时间

技巧 11　设置系统日期

本实例显示了设置系统日期的方法，运行本实例编译后的可执行文件，即可输入指定的年份、月份、日期。

实现思想：

1. 获取当前系统日期

获取当前系统日期使用函数 getdate()实现。该函数在 dos.h 头文件中，其语法如下：

```
void getdate (struct date *datep);
```

功能：将计算机内的日期写入 datep 指向的 date 结构中，以供用户使用。

date 结构如下：

```
struct date
{
    int da_year;                    /* Year - 1980 */
    char da_day;                    /* Day of the month */
    char da_mon;                    /* Month (1 = Jan) */
};
```

2. 设置系统日期

设置系统日期使用函数 setdate()实现。该函数在 dos.h 头文件中，其语法如下：

```
void setdate (struct date *datep);
```

功能：将计算机内的日期改成由 datep 指向的 date 结构所指定的日期。

例 14.11 应用 getdate()和 setdate()函数实现设置系统日期，实现步骤如下（光盘位置：光盘\MR\Instance\14\11）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdlib.h、dos.h。

```
#include<stdio.h>
#include<dos.h>
```

（3）设置“年”，代码如下：

```
printf("\n\nPlease set year ");          //提示指定年
gets(str);                               //获取控制台输入的字符串
setdt.da_year=atol(str);                 //将字符串转换为整数并设置年结构成员
```

（4）设置“月”，代码如下：

```
printf("\nPlease set month ");           //提示指定月
gets(str);                               //获取控制台输入的字符串
setdt.da_mon=atol(str);                  //将字符串转换为整数并设置月结构成员
```

（5）设置“日”，代码如下：

```
printf("\nPlease set day ");             //提示指定日
gets(str);                               //获取控制台输入的字符串
setdt.da_day=atol(str);                  //将字符串转换为整数并设置日结构成员
```

（6）设置系统日期，并显示设置后的系统日期，代码如下：

```
setdate(&setdt);                         //设置系统日期
getdate(&setdt);                         //获取系统日期
```

（7）主要程序代码如下：

```
main()
{
    char *str;                           //字符指针
    struct date  setdt;                  //声明 date 结构变量 setdt
```

```
    getdate(&setdt);                                          //获取当前系统日期
    //输出系统日期
    printf("\nSystem date %02d-%02d-%04d",setdt.da_mon,setdt.da_day,setdt.da_year);
    printf("\n\nPlease set year ");                          //提示设置年
    gets(str);                                                //从控制台获取字符串
    setdt.da_year=atol(str);                                  //将字符串转换为整数并设置年结构成员
    printf("\nPlease set month ");                           //提示指定月
    gets(str);                                                //从控制台获取字符串
    setdt.da_mon=atol(str);                                   //将字符串转换为整数并设置月结构成员
    printf("\nPlease set day ");                             //提示指定日
    gets(str);                                                //获取控制台输入的字符串
    setdt.da_day=atol(str);                                   //将字符串转换为整数并设置日结构成员
    setdate(&setdt);                                          //设置系统日期
    getdate(&setdt);                                          //获取系统日期
    //输出系统日期
    printf("\nAfter setting date %02d-%02d-%04d",setdt.da_mon,setdt.da_day,setdt.da_year);
}
```

运行本实例，效果如图 14.11 所示。

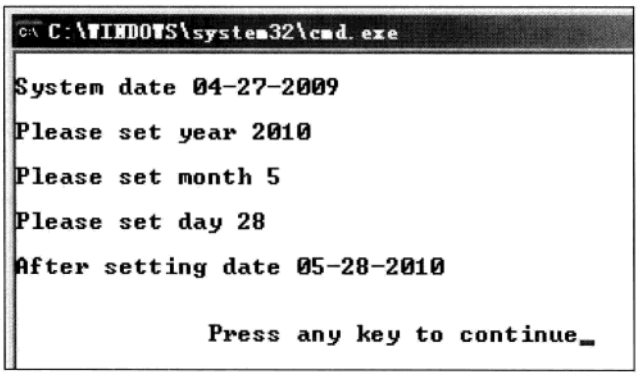


图 14.11　获得文件夹大小的效果

技巧 12　获取 BIOS 常规内存容量

计算机的地址空间为 1MB，其中低端 640KB 用作 RAM，供 DOS 及应用程序使用。低端的 640KB 称为常规内存。本实例用于介绍获取 BIOS 常规内存的方法。

实现思想：

获取 BIOS 常规内存容量通过使用函数 biosmemory()实现。该函数在 bios.h 头文件中，语法如下：

```
int biosmemory();
```

该函数的功能是返回常规内存大小，以 KB 为单位。

例 14.12　应用 biosmemory()函数实现获取 BIOS 常规内存容量，实现步骤如下（光盘位置：光盘\MR\Instance\14\12）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、bios.h。

```
#include <stdio.h>
#include <bios.h>
```

（3）主要程序代码如下：

```
main()
{
    int memsize;                                          //声明整型变量
    memsize=biosmemory();                                 //获取 BIOS 常规内存容量
    printf("\nBIOS regular memory size is %dKb",memsize); //输出 BIOS 常规内存容量值
    getch();
}
```

运行本实例，效果如图 14.12 所示。

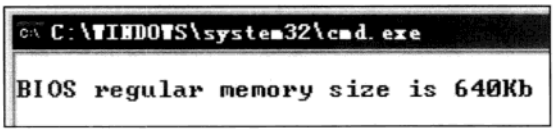


图 14.12　获取 BIOS 常规内存容量

技巧 13　获取 CMOS 密码

CMOS 是电脑主板上的一块可读写的 RAM 芯片，主要用来保存当前系统的硬件配置。为了防止他人随意更改硬件设置，就需要为 CMOS 设置密码。当计算机使用者将密码遗忘时，为了保全当前的硬件配置信息就不能进行清空 CMOS 的操作，而需要使用应用程序获取 CMOS 的密码。

实现思想：

AWARD 公司 BIOS 的 CMOS 密码存放在 CMOS 芯片 0x1C 和 0x1D 处的两个字节中，将这个两个字节的数据读取出来并用四进制表示，就是密码。读取 0x1C 和 0x1D 处的两个字节数据需要“字节写入指定的输出端口”、“从指定的输入端口读取字节”两个步骤。

1. 字节写入指定的输出端口

将一个字节写入输出端口通过使用函数 outportb 实现。该函数在 dos.h 头文件中，其语法如下：

```
void outportb(int port,char byte);
```

对其中参数进行以下说明。

- port：端口地址。
- byte：一字节。

2. 从指定的输入端口读取字节

要从指定的输入端口读取一个字节可以通过使用函数 inportb 实现。该函数在 dos.h 头文件中，其语法如下：

```
int inportb(int port);
```

该函数的功能是从指定的输入端口读入一个字节,并返回这个字节。

其中 port 参数指定了端口地址。

例 14.13　应用 outportd()和 inportd()函数实现获取 CMOS 密码，实现步骤如下（光盘位置：光盘\MR\Instance\14\13）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdlib.h、dos.h。

```
#include <stdio.h>
#include <dos.h>
```

（3）读取 0x1D 处数据并用四进制表示，代码如下：

```
outportb(0x70, 0x1d);                                   //在 0x70 地址写入字节 0x1d
comspass = inportb(0x71);                               //读取 0x71 地址的字节
//6 至 0 循环，步长为-2
for (n = 6; n >= 0; n -= 2)
{
    temp = comspass;                                    //记录 0x71 地址的字节内容
    temp >>= n;                                         //右移
    temp = temp & 0x03;                                 //与 0x03 与运算
    printf("%d", temp);                                 //以 4 进制的形式输出 0x1D 地址处的数据
}
```

（4）读取 0x1C 处数据并用四进制表示，代码如下：

```
outportb(0x70, 0x1c);                                   //在 0x70 地址写入字节 0x1c
result = inportb(0x71);                                 //读取 0x71 地址的字节
//6 至 0 循环，步长为-2
for (n = 6; n >= 0; n -= 2)
{
    temp = comspass;                                    //记录 0x71 地址的字节内容
    temp >>= n;                                         //右移
    temp = temp & 0x03;                                 //与 0x03 与运算
    printf("%d", temp);                                 //以四进制的形式输出 0x1C 地址处的数据
}
```

（5）主要程序代码如下：

```
void main() {
    int n;                                              //声明整型变量
    char comspass;                                      //字符类型变量
    char temp = 0;                                      //声明字符串类型变量初始值为 0
    int result;                                         //声明整型变量
    printf("\nThis program only is tested at win98 and award bios!!\n");        //输出字符串
```

```
    printf("The password of the cmos is:");
    outportb(0x70, 0x1d);                          //在 0x70 地址写入字节 0x1d
    comspass = inportb(0x71);                      //读取 0x71 地址的字节
    for (n = 6; n >= 0; n -= 2) {                  //6 至 0 循环，步长为-2
        temp = comspass;                           //记录 0x71 地址的字节内容
        temp >>= n;                                //右移
        temp = temp & 0x03;                        //与 0x03 与运算
        printf("%d", temp);                        //以四进制的形式输出 0x1D 地址处的数据
    }
    outportb(0x70, 0x1c);                          //在 0x70 地址写入字节 0x1c
    result = inportb(0x71);                        //读取 0x71 地址的字节
    for (n = 6; n >= 0; n -= 2) {                  //6 至 0 循环，步长为-2
        temp = comspass;                           //记录 0x71 地址的字节内容
        temp >>= n;                                //右移
        temp = temp & 0x03;                        //与 0x03 与运算
        printf("%d", temp);                        //以四进制的形式输出 0x1C 地址处的数据
    }
}
```

运行本实例，效果如图 14.13 所示。

```
This program only is tested at win98 and award bios!!
The password of the cmos is:00000000

            Press any key to continue_
```

图 14.13　获取 CMOS 密码

技巧 14　鼠标中断

使用鼠标是为了使计算机的操作更加简单快捷，代替键盘繁琐的指令。在 DOS 程序的开发过程中，如果要求应用程序可以使用鼠标进行相关的操作，那么就需要设置鼠标中断。本实例通过设置鼠标中断，实现了显示或隐藏鼠标的功能。

实现思想：

1. 初始化鼠标

鼠标初始化通过使用函数 int86()执行 0x33 号中断实现。该函数在头文件 dos.h 中，该语法如下：

```
int int86(int intr_num,union REGS *inregs,union REGS *outregs);
```

该函数的功能是执行 intr_num 号中断，用户定义的寄存器值存于结构 inregs 中， 执行完后将返回的寄存器值存于结构 outregs 中。

2. 显示或隐藏鼠标

显示或隐藏鼠标通过设置 REGS 联合体中的 x.ax 的值，配合使用函数 int86()实现。当 x.ax 的值为 1 时，显示鼠标；当 x.ax 的值为 2 时，隐藏鼠标。

REGS 联合体如下：

```
union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

其中的 WORDREGS 定义为：

```
struct WORDREGS
{
    unsigned int ax, bx, cx, dx, si, di,
    cflag                                   /*进位标志*/,
    flags                                   /*标志寄存器*/;
};
```

BYTEREGS 定义为：

```
struct BYTEREGS
{
    unsigned char al, ah, bl, bh, cl, ch, dl, dh;
};
```

例 14.14　通过设置 REGS 联合体中的 max 的值，应用函数 int86()实现显示鼠标和隐藏鼠标。实现步骤如下（光盘位置：光盘\MR\Instance\14\14）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 conio.h、stdlib.h、dos.h 和 stdio.h。

```
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <stdio.h>
```

（3）初始化鼠标，代码如下：

```
union REGS regs;                                    //定义寄存器型共同体
int mousedr;                                        //声明整型变量
clrscr();                                           //清屏
printf("Initializing Mouse...\n\n");                //输出字符串
regs.x.ax=0;                                        //设置 ax 值为 0
int86(0x33,&regs,&regs);
mousedr=regs.x.ax;                                  //记录 ax 值
if(mousedr==0)                                      //当值等于 0 时
{
    printf("initialize mouse error!");              //提示鼠标初始化失败
    exit(1);                                        //终止程序
}
printf("Finished initializing!\n\n");               //提示完成初始化
```

（4）显示鼠标，代码如下：

```
regs.x.ax=1;                                        //设置 ax 值为 1
int86(0x33,&regs,&regs);                            //显示鼠标
```

（5）隐藏鼠标，代码如下：

```
regs.x.ax=2;                                    //设置 ax 值为 1
int86(0x33,&regs,&regs);                        //隐藏鼠标
```

（6）主要程序代码。

```
int main()
{
    union REGS regs;                            //定义寄存器型共同体
    int mousedr;                                //声明整型变量
    clrscr();                                   //清屏
    printf("Initializing Mouse...\n\n");        //输出字符串
    regs.x.ax=0;                                //设置 ax 值为 0
    int86(0x33,&regs,&regs);
    mousedr=regs.x.ax;                          //记录 ax 值
    if(mousedr==0)                              //等于 0 时
    {
    printf("initialize mouse error!");          //提示初始化鼠标错误
    exit(1);                                    //终止程序
    }
    printf("Finished initializing!\n\n");       //完成初始化
    printf("Showing Mouse...\n");               //提示显示鼠标
    regs.x.ax=1;                                //设置 ax 值为 1
    int86(0x33,&regs,&regs);                    //显示鼠标
    printf("you can use the mouse.....\n\n");   //提示能用鼠标
    printf("Press any key to hide the mouse.\n");//提示按任意键隐藏鼠标
    getch();
    regs.x.ax=2;                                //设置 ax 值为 2
    int86(0x33,&regs,&regs);                    //隐藏鼠标
    printf("\nthe mouse is hided\n");           //提示鼠标正在隐藏
    getch();
    return 1;
}
```

运行本实例，效果如图 14.14 所示。

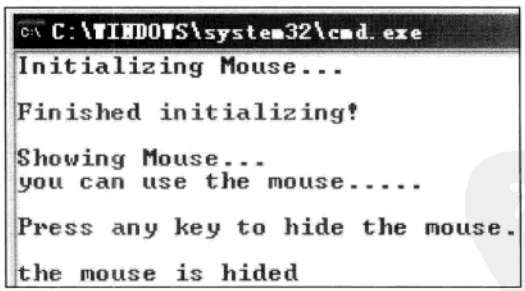


图 14.14　鼠标中断

技巧 15　备份/恢复硬盘分区表

由于病毒的破坏或用户的误操作可能导致硬盘的分区表被破坏，于是就需要重新创建分区表。重新创建分区表不但浪费时间，而且硬盘中的全部文件将会丢失。为了便于对分区表的修复，可以使用应用程序将分区表进行备份，当分区表被破坏时使用备份进行恢复。这样不但节约时间，而且

使硬盘中的数据得以保留。本技巧主要介绍了备份或恢复分区表的方法。

实现思想：

保存分区表时使用函数 biosdisk 对硬盘分区进行读写操作。bios 函数通过 INT 13H 中断向 BIOS 发出磁盘操作指令，从指定的扇区位置读取 512 字节数据并保存在 buffer 中。然后使用文件操作函数 fopen、fwrite 等进行保存。恢复分区表则调用 biosdisk 函数将保存的分区表文件的内容恢复到分区表中。

1. 读/写磁盘扇区

读/写磁盘扇区使用 biosdisk 函数实现。biosdisk 函数在 bios.h 中，其语法如下：

```
int biosdisk(int cmd,int drive,int head,int track, int sector,int nsects,void *buffer);
```

2. 文件操作

无论备份或恢复分区表都要使用函数 fopen、fwrite 函数配合函数 biosdisk 使用。当保存分区表时，使用函数 fopen 创建新文件，使用函数 fwrite 将 buffer 中的数据保存在文件中。当恢复分区表时首先使用 fopen 函数打开分区表文件，然后使用函数 fread 将文件中的数据保存在 buffer 中，最后使用函数 biosdisk 写入磁盘扇区中。函数 fopen、fwrite、fread 都在头文件 stdio.h 中。

函数 fopen 语法如下：

```
FILE *fopen(char *filename,char *type);
```

该函数的功能是打开一个文件 filename，打开方式为 type，并返回这个文件指针。

函数 fwrite 语法如下：

```
int fwrite(void *ptr,int size,int nitems,FILE *stream);
```

该函数的功能是向流 stream 中写入 nitems 个长度为 size 的字符串，字符串在 ptr 中。

函数 fread 语法如下：

```
int fread(void *ptr,int size,int nitems,FILE *stream);
```

该函数的功能是从流 stream 中读入 nitems 个长度为 size 的字符串存入 ptr 中。

例 14.15　使用 biosdisk、fopen、fwrite 函数对 C 盘的 0 磁头 0 柱面 1 扇区中的数据进行备份（光盘位置：光盘\MR\Instance\14\15）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、bios.h、fcntl.h、types.h 和 stat.h。

```
#include <stdio.h>
#include <bios.h>
```

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
```

（3）备份分区表

使用 biosdisk、fopen、fwrite 函数对 C 盘的 0 磁头 0 柱面 1 扇区中的数据进行备份，代码如下：

```
if (*argv[1] == 's' ||  *argv[1] == 'S')                          //当参数为大写或小写 S 时
{
    result = biosdisk(2, 0x80, 0, 0, 1, 1, buffer);               //读取分区表
    if (!result)                                                  //如果读取成功
    {
        printf(" Read partition table successfully!\n");          //显示分区表读取成功
        if ((fp = fopen("c:\\diskp.bak", "wb+")) == NULL)         //在 C 盘根目录下创建分区表备份文件
        {
            fprintf(stderr, " Can't creat file:c:\\diskp.bak \n"); //当文件创建失败时提示文件创建失败
            exit(1);                                              //终止当前程序
        }
        fwrite(buffer, 1, 512, fp);                               //当分区表数据写入备份文件
        fclose(fp);                                               //关闭文件
        printf(" Partition table save successfully!\n");          //显示分区表数据备份成功
        return 0;
    }
    else
    {
        fprintf(stderr, " Fail to read partition table!");        //显示分区表数据备份失败
        exit(1);
    }
}
```

（4）使用 biosdisk、fopen、fwrite 函数对 C 盘的 0 磁头 0 柱面 1 扇区中的数据进行恢复，代码如下：

```
if (*argv[1] == 'r' ||  *argv[1] == 'R')                          //当参数为大写或小写 R 时
{
    if ((fp = fopen("c:\\diskp.bak", "rb+")) == NULL)             //打开分区表备份文件
    {
        fprintf(stderr, " Can't open file!");                     //当打开失败时提示
        exit(1);                                                  //终止程序
    }
    fread(buffer, 1, 512, fp);                                    //读取备份文件中的分区表信息
    result = biosdisk(3, 0x80, 0, 0, 1, 1, buffer);               //将分区表数据写入硬盘
    if (!result)                                                  //如果分区表恢复成功
    {
        printf(" Partition table restore successfully!\n");       //提示分区表恢复成功
        fclose(fp);                                               //关闭文件
        return 0;
    }
    else                                                          //否则
    {
        fprintf(stderr, " Failt to restore partition table!");    //显示恢复分区表失败
        fclose(fp);                                               //关闭文件
        exit(1);                                                  //终止程序
    }
}
```

（5）主要代码如下：

```
int main(int argc, char *argv[])
{
    int result;                                                    //声明整型变量
    char buffer[512];                                              //声明字符类型数组
    FILE *fp;                                                      //文件指针
    if (argc == 1)                                                 //当参数 argc 值为 1 时
        info();                                                    //自定义函数用于显示程序说明
    if (*argv[1] == 's' ||  *argv[1] == 'S')                       //当字符参数为大写或小写 S 时
    {
        result = biosdisk(2, 0x80, 0, 0, 1, 1, buffer);            //读取分区表
        if (!result)                                               //如果读取成功
        {
            printf(" Read partition table successfully!\n");       //显示分区表读取成功
            if ((fp = fopen("c:\\diskp.bak", "wb+")) == NULL)      //在 C 盘根目录下创建分区表备份文件
            {
                fprintf(stderr, " Can't creat file:c:\\diskp.bak \n");//当文件创建失败时提示文件创建失败
                exit(1);                                           //终止程序
            }
            fwrite(buffer, 1, 512, fp);                            //当分区表数据写入备份文件
            fclose(fp);                                            //关闭文件
            printf(" Partition table save successfully!\n");       //显示分区表数据备份成功
            return 0;
        }
        else                                                       //否则
        {
            fprintf(stderr, " Fail to read partition table!");     //显示恢复分区表失败
            exit(1);                                               //终止程序
        }
    }
    if (*argv[1] == 'r' ||  *argv[1] == 'R')                       //当参数为大写或小写 R 时
    {
        if ((fp = fopen("c:\\diskp.bak", "rb+")) == NULL)          //打开分区表备份文件
        {
            fprintf(stderr, " Can't open file!");                  //当打开失败时提示
            exit(1);  //终止程序
        }
        fread(buffer, 1, 512, fp);                                 //读取备份文件中的分区表信息
        result = biosdisk(3, 0x80, 0, 0, 1, 1, buffer);            //将分区表数据写入硬盘
        if (!result)                                               //如果分区表恢复成功
        {
            printf(" Partition table restore successfully!\n");    //显示分区表数据备份成功
            fclose(fp);                                            //关闭文件
            return 0;
        }
        else
        {
            fprintf(stderr, " Failt to restore partition table!");//显示恢复分区表失败
            fclose(fp);                                            //关闭文件
            exit(1);                                               //终止程序
        }
    }
    printf("\n Press any key to quit...");                         //提示按任意键退出
    getch();
    return 0;
}
```

运行本实例，效果如图 14.15 所示。

```
C:\WINDOWS\system32\cmd.exe
The correction method using this program is : program S or program R
   S---save partition table to file diskp.bak in c disk
   R---restore partion table from file diskp.bak in c disk

C:\>pbak s
 Read partition table successfully!
 Partition table save successfully!

C:\>
```

图 14.15　备份/恢复硬盘分区表

技巧 16　获取系统配置信息

如果用户需要配置系统环境，那么首先要查看当前的系统配置信息。系统配置信息包括系统日期、设备号、驱动器类型等。本实例程序通过端口获取系统配置信息。

实现思想：

系统信息存放在CMOS存储器中，本实例程序通过向端口 0x70 发送一字节数据，并从端口 0x71 读取一个字节的数据，从而实现读取 CMOS 中信息的功能。

1. 字节写入指定的输出端口

将一个字节写入输出端口通过使用函数 outportb()实现。该函数在 dos.h 头文件中，其语法如下。

```
void outportb(int port,char byte);
```

参数如下。

port：端口地址。

byte：一字节。

2. 从指定的输入端口读取字节

从指定的输入端口读取一个字节通过使用函数 inportb()实现。该函数在 dos.h 头文件中，其语法如下。

```
int inportb(int port);
```

功能：

从指定的输入端口读入一个字节，并返回这个字节。

参数如下。

port：端口地址。

例 14.16　创建 SYSTEMINFO 结构，用于获取系统配置信息，具体步骤如下（光盘位置：光盘\MR\Instance\14\16）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、dos.h。

```
#include <stdio.h>
#include <dos.h>
```

（3）创建 SYSTEMINFO 结构。创建结构 SYSTEMINFO 用于获取系统配置信息，其创建代码如下。

```
struct SYSTEMINFO
{
    unsigned char current_second;                    //当前系统时间（秒）
    unsigned char alarm_second;                      //闹钟时间（秒）
    unsigned char current_minute;                    //当前系统时间（分）
    unsigned char alarm_minute;                      //闹钟时间（分）
    unsigned char current_hour;                      //当前系统时间（小时）
    unsigned char alarm_hour;                        //闹钟时间（小时）
    unsigned char current_day_of_week;               //当前系统时间（星期几）
    unsigned char current_day;                       //当前系统时间（日）
    unsigned char current_month;                     //当前系统时间（月）
    unsigned char current_year;                      //当前系统时间（年）
    unsigned char status_registers[4];               //寄存器状态
    unsigned char diagnostic_status;                 //诊断位
    unsigned char shutdown_code;                     //关机代码
    unsigned char drive_types;                       //驱动类型
    unsigned char reserved_x;                        //保留位
    unsigned char disk_1_type;                       //硬盘类型
    unsigned char reserved;                          //保留位
    unsigned char equipment;                         //设备号
    unsigned char lo_mem_base;
    unsigned char hi_mem_base;
    unsigned char hi_exp_base;
    unsigned char lo_exp_base;
    unsigned char fdisk_0_type;                      //软盘驱动器 0 类型
    unsigned char fdisk_1_type;                      //软盘驱动器 1 类型
    unsigned char reserved_2[19];                    //保留位
    unsigned char hi_check_sum;
    unsigned char lo_check_sum;
    unsigned char lo_actual_exp;
    unsigned char hi_actual_exp;
    unsigned char century;                           //世纪信息
    unsigned char information;
    unsigned char reserved3[12];                     //保留位
};
```

（4）为 SYSTEMINFO 结构变量赋值，获取系统配置信息，代码如下。

```
for(i=0;i<size;i++)
    {
        outportb(0x70,(char)i);                      //输出整数到硬件端口中
        byte=inportb(0x71);                          //从硬件端口中输入
        //以字节为单位依次为变量 SYSTEMINFO 赋值//
        *ptr_sysinfo++=byte;
    }
```

（5）主要程序代码。

```
int main()
{
    struct SYSTEMINFO systeminfo;                                    //声明 SYSTEMINFO 结构变量
    int i, size;                                                     //声明整型变量
    char *ptr_sysinfo, byte; //声明字符指针变量与字符变量
    clrscr();                                                        //清屏
    puts("*****************************************");
    puts("* Get configure information of your system *");
    puts("*****************************************");
    size = sizeof(systeminfo);                                       //结构占用字节数
    ptr_sysinfo = (char*) &systeminfo;                               //将结构地址转换为字符指针
    for (i = 0; i < size; i++)
    {
        outportb(0x70, (char)i);                                     //输出整数到硬件端口中
        byte = inportb(0x71);                                        //从硬件端口中输入
        //以字节为单位依次为变量 SYSTEMINFO 赋值
        *ptr_sysinfo++ = byte;
    }
    printf("--The hard disk type is: %d.\n", systeminfo.disk_1_type); //硬盘类型
    printf("--The drive type     is: %d.\n", systeminfo.drive_types); //驱动类型
    printf("--The equipment_num  is: %d.\n", systeminfo.equipment);   //设备号
    printf("--The current date   is: %x/%x/0%x.\n", systeminfo.current_day,
        systeminfo.current_month, systeminfo.current_year);          //当前日期
    printf("--The current time   is: %x:%x:%x.\n", systeminfo.current_hour,
        systeminfo.current_minute, systeminfo.current_second);       //当前时间
    printf("--The alarm time     is: %x:%x:%x.\n", systeminfo.alarm_hour,
        systeminfo.alarm_minute, systeminfo.alarm_second);           //报警时间
    getch();
    return 0;
}
```

运行本实例编译后的可执行文件，可以对当前系统配置信息进行显示，显示信息如图 14.16 所示。

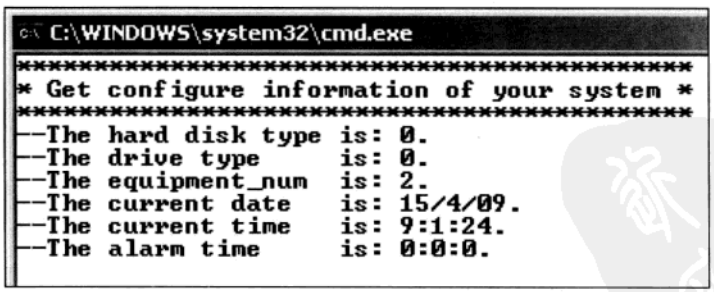


图 14.16　获取系统配置信息

技巧 17　恢复内存文本

用户在编写文本时，如果遇到系统异常中断，则会导致正在编写的文本丢失。本实例通过恢复内存文本的方式，来介绍抢救编写的文本的方法。

实现思想：

本实例程序使用内存读字符函数 peekb()从内存中读取字符，并将获取到的字符输出到新建的文本文件中。

peekb 函数语法如下。

```
char peekb(int segment,unsigned offset);
```

功能：返回地址为 segment：offset 处字节单元的值。

参数如下。

segment：段地址。

offset：偏移地址。

例 14.17　新建一个文件，并以二进制形式的操作。实现恢复内存文本，具体步骤如下（光盘位置：光盘\MR\Instance\14\17）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、dos.h。

```
#include<stdio.h>
#include<dos.h>
```

（3）新建一个文件，并以二进制形式的操作。代码如下。

```
fp=fopen(filename,"w+b");
```

（4）忽略 ASCII 码值小于 32 的字符和 ASCII 码值介于 126～160 之间的字符，以及半个汉字，代码如下。

```
//忽略 ASCII 码值小于 32 的字符和 ASCII 码值介于 126～160 之间的字符
if((ch<32&&ch!=13)||(ch>126&&ch<160))
{
    flag=0;
    continue;
}
if(ch>160&&flag==0)                                    //忽略半个汉字
{
    flag=ch;
    continue;
}
```

（5）将从内存中获取的字符输出到指定文件中。字符的输出使用 fputc()函数将字符写入流中。字符写入文件的代码如下。

```
if(ch>160&&flag==0)                                    //忽略半个汉字
{
    flag=ch;
```

```
    continue;
}
k++;
if(ch<160)                                          //判断 ch 小于 160
{
    fputc(ch,fp);
    if(ch==13)                                      //当 ch 等于 13 时执行下面“{}”内代码
    {
        fputc(10,fp);
        k=0;
    }
}
else
{
    fputc(flag,fp);
    fputc(ch,fp);
    k++;
    flag=0;
}
/*添加换行符*/
if(k==MAXLINE-1)
{
    fputc(13,fp);
    fputc(10,fp);
    k=0;
}
```

（6）输出字符后将文件关闭，关闭文件通过使用 fclose()函数实现。关闭文件的代码如下。

```
fclose(fp);
```

（7）主要程序代码如下。

```
int main()
{
    char *filename[32];                                 /*定义字符型数组*/
    FILE *fp;                                           /*声明指针*/
    char ch,flag;                                       /*定义字符型变量*/
    unsigned long n,m,k=0;
    clrscr();
    printf("Please input the object file:\n");          /*输出提示信息*/
    gets(filename);
    printf(".....");
    if((fp=fopen(filename,"w+b"))==0)
    {
        printf("Cannot open the file %s\n",filename);
        exit(0);
    }
    for(m=0;m<40960;m+=4096)
    {
        for(n=0;n<65536;n++)
        {
            ch=peekb(m,n);
            /*忽略 ASCII 码值小于 32 的字符和 ASCII 码值介于 126～160 之间的字符*/
            if((ch<32&&ch!=13)||(ch>126&&ch<160))
            {
                flag=0;
                continue;
            }
            /*忽略半个汉字*/
            if(ch>160&&flag==0)
```

```
            {
                flag=ch;
                continue;
            }
            k++;
            if(ch<160)
            {
                fputc(ch,fp);
                if(ch==13)
                {
                    fputc(10,fp);
                    k=0;
                }
            }
            else
            {
                fputc(flag,fp);
                fputc(ch,fp);
                k++;
                flag=0;
            }
            /*添加换行符*/
                if(k==MAXLINE-1)
                {
                    fputc(13,fp);
                    fputc(10,fp);
                    k=0;
                }
        }
    }
    printf("\nRecover the text successfully!!\n");
    fclose(fp);
    printf("\nPlease input any key to quit..\n");
    getch();
    return 0;
}
```

运行本实例编译后的可执行文件，可以将用户编写的文本从内存中挽救回来，程序运行结果如图 14.17 所示。

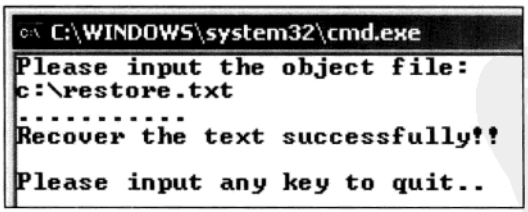


图 14.17　恢复内存文本

技巧 18　控制扬声器声音

在程序开发过程中，可能需要控制扬声器的声音，将发出的声音作为程序操作的提示音。本技巧主要用于介绍控制扬声器声音的方法。

实现技巧：

控制扬声器的声音通过使用 sound()函数实现。该函数在 dos.h 头文件中，其语法如下。

```
void sound(unsigned frequency);
```

功能：让扬声器发出指定频率的声音。

参数如下：

frequency：扬声器发声的频率。

例 14.18 在循环体内部使用 sound()函数控制扬声器发出指定频率的声音（光盘位置：光盘\MR\Instance\14\18）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 dos.h。

```
#include <dos.h>
```

（3）创建一循环体。

（4）在循环体内部使用 sound()函数控制扬声器发出指定频率的声音。

（5）主要代码如下。

```
main(void)
{
    unsigned fre=50;                          //声明无符号基本型变量
    int times;                                //声明整型变量
    for(times=0;times<1000;times++)           //0~999 循环
    {
        fre=(fre+times)%40000;                //生成声音频率
        sound(fre);                           //发出声音
        delay(1000);                          //延时 1 秒
    }
    nosound();                                //停止发声
}
```

5

第 5 篇
算法和效率
——程序员的得力助手

第 15 章　常用的 30 个算法及应用



第 15 章　常用的 30 个算法及应用

程序由两部分组成：算法和数据。算法是程序的灵魂，我们在编写程序的过程中无时无刻不在应用算法，可以说精通算法是一名优秀程序员必须具备的素质。本章列出几种最常用的算法和一些经典的算法应用供读者朋友们学习。本章的主要内容包括以下几个方面。

- 排序算法
- 查找算法
- 猜想与定理
- 数据结构
- 加密算法
- 算法应用

算法 01　冒泡排序

实例说明

冒泡排序是一种最常用的排序方法，其过程很简单，就像气泡一样越往上走越大，因此被人们形象地称为冒泡排序法。本实例用冒泡法对任意输入的 10 个数进行由小到大排序。运行结果如图 15.1 所示。

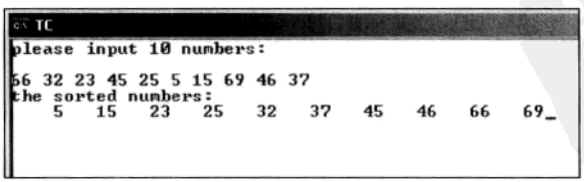


图 15.1　冒泡排序

技术要点

本实例要求用冒泡法对 10 个数进行由小到大排序，下面就介绍下冒泡法的基本思路。

如果要对 n 个数进行冒泡排序，那么要进行 $n-1$ 轮比较，在第 1 轮比较中要进行 $n-1$ 次两两比较，在第 j 趟比较中要进行 $n-j$ 次两两比较。从这个基本思路中我们也会发现，轮数决定了两两比较的次数，这样我们就很容易将两个 for 循环联系起来了。

实现过程

例 15.1　冒泡排序算法（光盘位置：光盘\MR\Instance\15\01）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）通过两个 for 循环实现冒泡排序的全过程，外层 for 循环决定冒泡排序的趟数，内层 for 循环决定每趟所进行两两比较的次数。

（4）主要程序代码如下。

```
main()
{
    int i, j, t, a[11];                                  /*定义变量及数组为基本整型*/
    printf("please input 10 numbers:\n");
    for (i = 1; i < 11; i++)
        scanf("%d", &a[i]);                              /*从键盘中输入 10 个数*/
    for (i = 1; i < 10; i++)                             /*变量 i 代表比较的趟数*/
        for (j = 1; j < 11-i; j++)                       /*变量 j 代表每趟两两比较的次数*/
    if (a[j] > a[j + 1])
    {
        t = a[j];                                        /*利用中间变量实现两值互换*/
        a[j] = a[j + 1];
        a[j + 1] = t;
    }
    printf("the sorted numbers:\n");
    for (i = 1; i <= 10; i++)
        printf("%5d", a[i]);                             /*将冒泡排序后的顺序输出*/
}
```

举一反三

根据本实例，读者可以：

- 使用冒泡排序法将如下数字由小到大进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357；
- 使用冒泡排序法将如下数字由大到小进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357。

算法 02 选择排序

实例说明

选择排序的基本思想是：每一趟从待排序的记录中选取关键字最小的记录，顺序放在已排好序的序列最后，直到全部记录排序完毕。

本实例用选择排序法对以下数据进行由小到大排序，数据分别为 526、36、2、369、56、45、78、92、125、52。运行结果如图 15.2 所示。

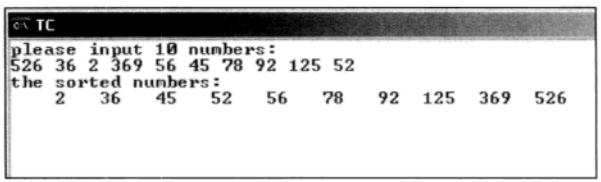


图 15.2 选择排序

技术要点

选择排序的基本算法是从待排序的区间中经过选择和交换后选出最小的数值存放到 a[0]中，再从剩余的未排序区间中经过选择和交换后选出最小的数值存放到 a[1]中，a[1]中的数字仅大于 a[0]，依此类推，即可实现排序。

实现过程

例 15.2 选择排序算法（光盘位置：光盘\MR\Instance\15\02）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）程序中用到两个 for 循环语句，第一个 for 循环是确定位置的，该位置是存放每次从待排序数列中经选择和交换后所选出的最小数。第二个 for 循环实现确定位置上的数与后面待排序区间中的数进行比较。

（4）主要程序代码。

```
main()
{
```

```
    int i, j, t, a[11];                          /*定义变量及数组为基本整型*/
    printf("please input 10 numbers:\n");
    for (i = 1; i < 11; i++)
        scanf("%d", &a[i]);                      /*从键盘中输入要排序的 10 个数字*/
    for (i = 1; i <= 9; i++)
        for (j = i + 1; j <= 10; j++)
            if (a[i] > a[j])                     /*如果后一个数比前一个数大则利用中间变量 t 实现两值互换*/
    {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    printf("the sorted numbers:\n");
    for (i = 1; i <= 10; i++)
        printf("%5d", a[i]);                     /*将排好序的数组输出*/
}
```

举一反三

根据本实例，读者可以：

- 使用选择排序法将如下数字由小到大进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357；
- 使用选择排序法将如下数字由大到小进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357。

算法 03　直接插入排序

实例说明

插入排序是把一个记录插入到已排序的有序序列中去，使整个序列在插入了该记录后仍然有序。插入排序中较简单的一种方法便是直接插入排序，其插入位置是通过将待插入的记录与有序区中的各记录自右向左依次比较其关键字值大小来确定的。本实例要求使用直接插入排序法将如下数字由小到大进行排序，数字分别为 25、12、36、45、2、9、39、22、98、37。运行结果如图 15.3 所示。

```
c:\ TC
please input number:
25 12 36 45 2 9 39 22 98 37
the original order:
   25   12   36   45    2    9   39   22   98   37
the sorted numbers:
    2    9   12   22   25   36   37   39   45   98
```

图 15.3　直接插入排序

技术要点

本实例算法过程如下。

原始顺序：25　12　36　45　2　9　39　22　98　37

趟　数	监 视 哨	排 序 结 果
1	12	(12，25，) 36，45，2，9，39，22，98，37
2	36	(12，25，36，) 45，2，9，39，22，98，37
3	45	(12，25，36，45，) 2，9，39，22，98，37
4	2	(2，12，25，36，45，) 9，39，22，98，37
5	9	(2，9，12，25，36，45，) 39，22，98，37
6	39	(2，9，12，25，36，39，45，) 22，98，37
7	22	(2，9，12，22，25，36，39，45，) 98，37
8	98	(2，9，12，22，25，36，39，45，98，) 37
9	37	(2，9，12，22，25，36，37，39，45，98)

注意：本算法中使用了监视哨，其目的是为了避免数据在后移时丢失。

实现过程

例 15.3　直接插入算法（光盘位置：光盘\MR\Instance\15\03）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义 isort（）函数实现直接插入排序，代码如下。

```
void insort(int s[], int n)                    /*自定义函数 isort*/
{
    int i, j;
    for (i = 2; i <= n; i++)                   /*数组下标从 2 开始，0 做监视哨，1 一个数据无可比性*/
    {
        s[0] = s[i];                           /*给监视哨赋值*/
        j = i - 1;                             /*确定要进行比较的元素的最右边位置*/
        while (s[0] < s[j])
        {
            s[j + 1] = s[j];                   /*数据右移*/
            j--;                               /*移向左边一个未比较的数*/
        }
        s[j + 1] = s[0];                       /*在确定的位置插入 s[i]*/
    }
}
```

（4）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int a[11], i;                                    /*定义数组及变量为基本整型*/
    printf("please input number:\n");
    for (i = 1; i <= 10; i++)
        scanf("%d", &a[i]);                          /*接收从键盘中输入的 10 个数据到数组 a 中*/
    printf("the original order:\n");
    for (i = 1; i < 11; i++)
        printf("%5d", a[i]);                         /*将为排序前的顺序输出*/
    insort(a, 10);                                   /*调用自定义函数 isort()*/
    printf("\nthe sorted numbers:\n");
    for (i = 1; i < 11; i++)
        printf("%5d", a[i]);                         /*将排序后的数组输出*/
    printf("\n");
}
```

举一反三

根据本实例，读者可以：

- 使用直接插入排序法将如下数字由小到大进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357；
- 使用直接插入排序法将如下数字由大到小进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357。

算法 04　归并排序

实例说明

用归并排序法对以下数据进行由小到大排序，数据分别为 695、458、362、789、12、15、163、23、2、986。运行结果如图 15.4 所示。

```
C:\ TC
please input 10 numbers:
695 458 362 789 12 15 163 23 2 986
the sorted numbers:
    2   12   15   23  163  362  458  695  789  986
```

图 15.4　归并排序

技术要点

归并是将两个或多个有序记录序列合并成一个有序序列。归并方法有多种，一次对两个有序记录序列进行归并，称为二路归并排序。也有三路归并排序及多路归并排序。本实例是二路归并排序，基本方法如下。

（1）将 n 个记录看成是 n 个长度为 1 的有序子表。

（2）将两两相邻的有序子表进行归并。

（3）重复执行（2），直到归并成一个长度为 n 的有序表。

实现过程

例 15.4 归并排序算法（光盘位置：光盘\MR\Instance\15\04）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义函数 merge（）函数，实现一次归并排序，代码如下。

```
void merge(int r[], int s[], int x1, int x2, int x3)  /*实现一次归并排序函数*/
{
    int i, j, k;
    i = x1;                                           /*第一部分的开始位置*/
    j = x2 + 1;                                       /*第二部分的开始位置*/
    k = x1;
    while ((i <= x2) && (j <= x3))
                                                      /*当 i 和 j 都在两个要合并的部分中*/
        if (r[i] <= r[j])
                                                      /*筛选两部分中较小的元素放到数组 s 中*/
    {
        s[k] = r[i];
        i++;
        k++;
    }
    else
    {
        s[k] = r[j];
        j++;
        k++;
    }
    while (i <= x2)
                                                      /*将 x1~x2 范围内的未比较的数顺次加到数组 r 中*/
        s[k++] = r[i++];
    while (j <= x3)
                                                      /*将 x2+1~x3 范围内的未比较的数顺次加到数组 r 中*/
        s[k++] = r[j++];
}
```

（4）自定义 merge_sort()函数，实现归并排序，代码如下。

```
void merge_sort(int r[], int s[], int m, int n)
{
    int p;
    int t[20];
    if (m == n)
```

```
        s[m] = r[m];
    else
    {
        p = (m + n) / 2;
        merge_sort(r, t, m, p);
        /*递归调用 merge_sort 函数将 r[m]~r[p]归并成有序的 t[m]~t[p]*/
        merge_sort(r, t, p + 1, n);    /*递归调用 merge_sort 函数将 r[p+1]~r[n]归并成有序的 t[p+1]~t[n]*/
        merge(t, s, m, p, n);          /*调用函数将前两部分归并到 s[m]~s[n]*/
    }
}
```

（5）主函数程序代码如下。

```
main()
{
    int a[11];
    int i;
    printf("please input 10 numbers:\n");
    for (i = 1; i <= 10; i++)
        scanf("%d", &a[i]);
    /*从键盘中输入 10 个数*/
    merge_sort(a, a, 1, 10);           /*调用 merge_sort 函数进行归并排序*/
    printf("the sorted numbers:\n");
    for (i = 1; i <= 10; i++)
        printf("%5d", a[i]);           /*将排序后的结构输出*/
}
```

举一反三

根据本实例，读者可以：

- 使用归并排序法将如下数字由小到大进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357；
- 使用归并排序法将如下数字由大到小进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357。

算法 05　希尔排序

链表是一种线性数据结构，使用它能动态地存储一种结构数据。链表是 n 个数据元素的有限序列，其元素需要自己定义，它既可以是一个整数或一个字符串，也可以是一个复杂的记录类型。在计算机程序设计中，经常使用链表，本节将详细介绍有关链表的相关知识。

实例说明

用希尔排序法对下面一组数字进行由小到大排序，数字分别为 69、56、12、136、3、55、46、99、88、25。运行结果如图 15.5 所示。

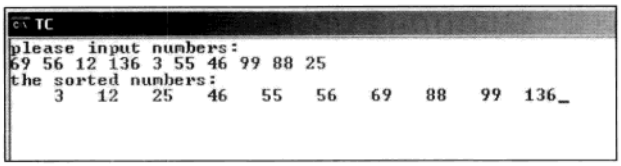


图 15.5 希尔排序

技术要点

希尔排序是在直接插入排序的基础上进行的改进，也就是将要排序的序列按固定增量分成若干组，等距离者在同一组中，然后再在组内进行直接插入排序。这里面的固定增量从 $n/2$ 开始，以后每次缩小到原来的一半。

实现过程

例 15.5 希尔排序算法（光盘位置：光盘\MR\Instance\15\05）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义 shsort()函数实现希尔排序，代码如下。

```
void shsort(int s[], int n)                          /*自定义函数 shsort*/
{
    int i, j, d;
    d = n / 2;                                       /*确定固定增量值*/
    while (d >= 1)
    {
        for (i = d + 1; i <= n; i++)                 /*数组下标从 d+1 开始进行直接插入排序*/
        {
            s[0] = s[i];                             /*设置监视哨*/
            j = i - d;                               /*确定要进行比较的元素的最右边位置*/
            while ((j > 0) && (s[0] < s[j]))
            {
                s[j + d] = s[j];                     /*数据右移*/
                j = j - d;                           /*向左移 d 个位置*/
            }
            s[j + d] = s[0];                         /*在确定的位置插入 s[i]*/
        }
        d = d / 2;                                   /*增量变为原来的一半*/
    }
}
```

（4）main()函数作为程序的入口函数，代码如下。

```
main()
{
```

```
    int a[11], i;                                    /*定义数组及变量为基本整型*/
    printf("please input numbers:\n");
    for (i = 1; i <= 10; i++)
        scanf("%d", &a[i]);                          /*从键盘中输入 10 个数据*/
    shsort(a, 10);                                   /*调用 shsort()函数*/
    printf("the sorted numbers:\n");
    for (i = 1; i <= 10; i++)
        printf("%5d", a[i]);                         /*将排好序的数组输出*/
}
```

举一反三

根据本实例，读者可以：

- 使用希尔排序法将如下数字由小到大进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357；
- 使用希尔排序法将如下数字由大到小进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357。

算法 06　快速排序

实例说明

用快速排序法对以下数据进行由小到大排序，数据分别为 99、45、12、36、69、22、62、796、4、696。运行结果如图 15.6 所示。

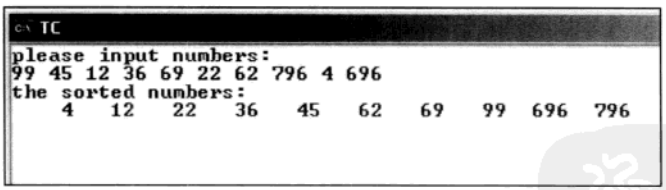


图 15.6　快速排序

技术要点

快速排序是冒泡排序的一种改进，主要的算法思想是在待排序的 n 个数据中取第一个数据作为基准值，将所有的记录分为 3 组，使得第一组中各数据值均小于或等于基准值，第二组便是做基准值的数据，第三组中各数据值均大于或等于基准值。这便实现了第一趟分割，然后再对第一组和第三组分别重复上述方法，依次类推，直到每组中只有一个记录为止。

实现过程

例 15.6 快速排序算法（光盘位置：光盘\MR\Instance\15\06）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义 qusort()函数实现希尔排序，代码如下。

```
void qusort(int s[], int start, int end)                    /*自定义函数 qusort()*/
{
    int i, j;                                               /*定义变量为基本整型*/
    i = start;                                              /*将每组首个元素赋给 i*/
    j = end;                                                /*将每组末尾元素赋给 j*/
    s[0] = s[start];                                        /*设置基准值*/
    while (i < j)
    {
        while (i < j && s[0] < s[j])
            j--;                                            /*位置左移*/
        if (i < j)
        {
            s[i] = s[j];                                    /*将 s[j]放到 s[i]的位置上*/
            i++;                                            /*位置右移*/
        }
        while (i < j && s[i] <= s[0])
            i++;                                            /*位置右移*/
        if (i < j)
        {
            s[j] = s[i];                                    /*将大于基准值的 s[j]放到 s[i]位置*/
            j--;                                            /*位置右移*/
        }
    }
    s[i] = s[0];                                            /*将基准值放入指定位置*/
    if (start < i)
        qusort(s, start, j - 1);                            /*对分割出的部分递归调用函数 qusort()*/
    if (i < end)
        qusort(s, j + 1, end);
}
```

（4）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int a[11], i;                                           /*定义数组及变量为基本整型*/
    printf("please input numbers:\n");
    for (i = 1; i <= 10; i++)
        scanf("%d", &a[i]);                                 /*从键盘中输入 10 个要进行排序的数*/
    qusort(a, 1, 10);                                       /*调用 qusort()函数进行排序*/
    printf("the sorted numbers:\n");
    for (i = 1; i <= 10; i++)
        printf("%5d", a[i]);                                /*输出排好序的数组*/
}
```

举一反三

根据本实例，读者可以：

- 使用快速排序法将如下数字由小到大进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357；
- 使用快速排序法将如下数字由大到小进行排序，数字分别为 45、68、12、32、7、85、456、258、123、357。

算法 07　顺序查找

实例说明

采用顺序查找法在关键字序列 25、65、356、569、55、65、45、66、154、66 中查找关键字为 154 的元素。运行结果如图 15.7 所示。

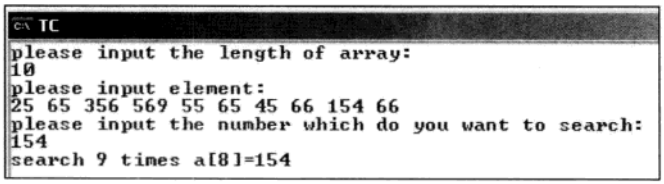


图 15.7　顺序查找

技术要点

顺序查找的算法思想是：从数组中的第一个元素 25 开始，将给定的值 154 与数组中的元素逐个比较，直到两者相等，查到所要找的元素并输出查找次数及该元素在数组中的位置。否则输出 no found 表示数组中没有要找的元素。

实现过程

例 15.7　顺序查找（光盘位置：光盘\MR\Instance\15\07）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义 search()函数实现顺序查找，代码如下。

```
void search(int key, int a[], int n)                    /*自定义函数 search*/
{
    int i, count = 0, count1 = 0;
    for (i = 0; i < n; i++)
    {
        count++;                                        /*count 记录查找次数*/
        if (a[i] == key)                                /*判断要查找的关键字与数组中的元素是否相等*/
        {
            /*输出查找次数及在数组中的位置*/
            printf("search %d times a[%d]=%d\n", count, i, key);
            count1++;                                   /*count1 记录查找成功次数*/
        }
    }
    if (count1 == 0)                                    /*判断是否查找到 h*/
        printf("no found!");                            /*如果未查找到输出 no found*/
}
```

（4）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int n, key, a[100], i;
    printf("please input the length of array:\n");
    scanf("%d", &n);                                    /*输入要输入的元素个数*/
    printf("please input element:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);                             /*输入元素存到数组 a 中*/
    printf("please input the number which do you want to search:\n");
    scanf("%d", &key);                                  /*指定要查找的元素*/
    search(key, a, n);                                  /*调用自定义的 search 函数*/
}
```

举一反三

根据本实例，读者可以：

- 采用顺序查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 146 的元素；
- 采用顺序查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 63 的元素。

算法 08 二分查找

实例说明

采用二分查找法在有序表 11、13、18、28、39、56、69、89、98、122 中查找关键字为 89 的

元素。运行结果如图 15.8 所示。

```
TC
please input the length of array:
10
please input the element:
11 13 18 28 39 56 69 89 98 122
please input the number which do you want to search:
89
success!
search 3 times!a[7]=89
```

图 15.8　顺序查找

技术要点

二分查找就是常说的折半查找，其基本思想是：首先选取表中间位置的记录，将其关键字与给定关键字 key 进行比较，若相等，则查找成功；若 key 值比该关键字值大，则要找的元素一定在右子表中，则继续对右子表进行折半查找；若 key 值比该关键字值小，则要找的元素一定在左子表中，继续对左子表进行折半查找。如此递推，直到查找成功或查找失败（查找范围为 0）。

实现过程

例 15.8　二分查找算法（光盘位置：光盘\MR\Instance\15\08）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义 binary_search()函数实现二分查找，代码如下。

```
void binary_search(int key, int a[], int n) /*自定义函数 binary_search*/
{
    int low, high, mid, count = 0, count1 = 0;
    low = 0;
    high = n - 1;
    while (low < high)                          /*当查找范围不为 0 时执行循环体语句*/
    {
        count++;                                /*count 记录查找次数*/
        mid = (low + high) / 2;                 /*求出中间位置*/
        if (key < a[mid])                       /*当 key 小于中间值*/
            high = mid - 1;                     /*确定左子表范围*/
        else if (key > a[mid])                  /*当 key 大于中间值*/
            low = mid + 1;                      /*确定右子表范围*/
        else if (key == a[mid])                 /*当 key 等于中间值证明查找成功*/
        {
            /*输出查找次数及所查找元素在数组中的位置*/
            printf("success!\nsearch %d times!a[%d]=%d", count, mid, key);
            count1++;                           /*count1 记录查找成功次数*/
            break;
        }
```

```
    }
    if (count1 == 0)                                    /*判断是否查找失败*/
        printf("no found!");                            /*查找失败输出 no found*/
}
```

（4）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int i, key, a[100], n;
    printf("please input the length of array:\n");
    scanf("%d", &n);                                    /*输入数组元素个数*/
    printf("please input the element:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);                             /*输入有序数列到数组 a 中*/
    printf("please input the number which do you want to search:\n");
    scanf("%d", &key);                                  /*输入要查找的关键字*/
    binary_search(key, a, n);                           /*调用自定义函数*/
}
```

举一反三

根据本实例，读者可以：

- 采用二分查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 146 的元素；
- 采用二分查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 63 的元素。

算法 09 分块查找

实例说明

采用分块查找法在有序表 11、13、18、28、39、56、69、89、98、122、135、146、156、256、289 中查找关键字为 89 的元素。运行结果如图 15.9 所示。

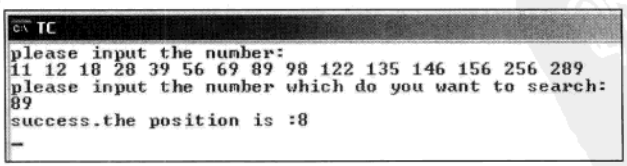


图 15.9 分块查找

技术要点

分块查找也称为索引顺序查找，要求将待查的元素均匀地分成块，块间按大小排序，块内不排

序，所以要建立一个块的最大（或最小）关键字表，称为索引表。

本实例中将给出的 15 个数按关键字大小分成了 3 块，这 15 数的排列是一个有序序列，也可以给出无序序列，但是也必须要满足分在第一块中的任意数都小于第二块中的所有数，第二块中的所有数都小于第三块中的所有数。当要查找关键字为 key 的元素时，则先用顺序查找在已建好的索引表中查出 key 所在的块中，再在对应的块中顺序查找 key，若 key 存在则输出其相应位置，否则输出提示信息。

实现过程

例 15.9　分块查找算法（光盘位置：光盘\MR\Instance\15\09）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）定义结构体 index，用于存储块的结构，并定义该结构体数组 index_table。

```
struct index                                        /*定义块的结构*/
{
    int key;
    int start;
    int end;
} index_table[4];                                   /*定义结构体数组*/
```

（4）自定义 block_search()函数，实现分块查找，代码如下。

```
int block_search(int key, int a[])                  /*自定义实现分块查找*/
{
    int i, j;
    i = 1;
    while (i <= 3 && key > index_table[i].key)      /*确定在哪个块中*/
        i++;
    if (i > 3)                                      /*大于分得的块数，则返回 0*/
        return 0;
    j = index_table[i].start;                       /*j 等于块范围的起始值*/
    while (j <= index_table[i].end && a[j] != key)  /*在确定的块内进行查找*/
        j++;
    if (j > index_table[i].end)          /*如果大于块范围的结束值，则说明没有要查找的数，j 置 0*/
        j = 0;
    return j;
}
```

（5）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int i, j = 0, k, key, a[16];
    printf("please input the number:\n");
    for (i = 1; i < 16; i++)
```

```
        scanf("%d", &a[i]);                                    /*输入由小到大的 15 个数*/
    for (i = 1; i <= 3; i++)
    {
        index_table[i].start = j + 1;                          /*确定每个块范围的起始值*/
        j = j + 1;
        index_table[i].end = j + 4;                            /*确定每个块范围的结束值*/
        j = j + 4;
        index_table[i].key = a[j];                             /*确定每个块范围中元素的最大值*/
    }
    printf("please input the number which do you want to search:\n");
    scanf("%d", &key);                                         /*输入要查询的数值*/
    k = block_search(key, a);                                  /*调用函数进行查找*/
    if (k != 0)
        printf("success.the position is :%d\n", k);            /*如果找到该数，则输出其位置*/
    else
        printf("no found!");                                   /*若未找到则输出提示信息*/
}
```

举一反三

根据本实例，读者可以：

- 采用分块查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 146 的元素；
- 采用分块查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 63 的元素。

算法 10 哈希查找

实例说明

编程实现哈希查找，要求如下：已知哈希表长度为 11，哈希函数为 H（key）=key%11，随机产生待散列的小于 50 的 8 个元素，同时采用线性探测再散列的方法处理冲突，任意输入要查找的数据，无论找到与否均给出提示信息。运行结果如图 15.10 所示。

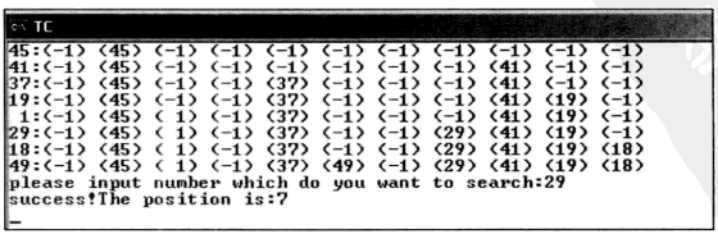


图 15.10 哈希查找

技术要点

哈希函数的构造方法常用的有 5 种，分别是数字分析法、平方取中法、分段叠加、伪随机数法、余数法，其中余数法比较常用。因为本实例中已给出哈希函数，所以不必构造，直接按照题中给的哈希函数来运算即可。

虽然通过构造好的哈希函数可以减少冲突，但是冲突是不可能完全避免的，所以就相应地产生了避免哈希冲突的常用的 4 种方法，如下所示。

1．开放定址法。

- 线性探测再散列。
- 二次探测再散列。

2．链地址法。

3．再哈希法。

4．建立公共溢出区。

开放定址法中的线性探测再散列比较常用，该方法的特点是冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

实现过程

例 15.10　哈希查找（光盘位置：光盘\MR\Instance\15\10）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，进行宏定义并声明全局变量。

```
#include <stdio.h>
#include <time.h>
#define Max 11
#define N 8
int hashtable[Max];
```

（3）自定义 func()函数，用于哈希函数的值。

```
int func(int value)
{
    return value % Max;                          /*哈希函数*/
}
```

（4）自定义 search()函数，实现哈希查找，代码如下。

```
int search(int key)                                    /*自定义函数实现哈希查询*/
{
    int pos, t;
    pos = func(key);                                   /*哈希函数确定出的位置*/
    t = pos;                                           /*t 存放确定出的位置*/
    while (hashtable[t] != key && hashtable[t] !=  - 1)
                                                       /*如果该位置上不等于要查找的关键字且不为空*/
    {
        t = (t + 1) % Max;                             /*利用线性探测求出下一个位置*/
        if (pos == t)
            /*如果经多次探测又回到原来用哈希函数求出的位置则说明要查找的数不存在*/
            return  - 1;
    }
    if (hashtable[t] ==  - 1)                          /*如果探测的位置是-1 则说明要查找的数不存在*/
        return NULL;
    else
        return t;
}
```

（5）自定义 creathash()函数，实现哈希表创建，代码如下。

```
void creathash(int key)                                /*自定义函数创建哈希表*/
{
    int pos, t;
    pos = func(key);                                   /*哈希函数确定元素的位置*/
    t = pos;
    while (hashtable[t] !=  - 1)
                                                       /*如果该位置有元素存在则进行线性探测再散列*/
    {
        t = (t + 1) % Max;
        if (pos == t)
        /*如果冲突处理后确定的位置与原位置相同则说明哈希表已满*/
        {
            printf("hash table is full\n");
            return ;
        }
    }
    hashtable[t] = key;                                /*将元素放入确定的位置*/
}
```

（6）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int flag[50];
    int i, j, t;
    for (i = 0; i < Max; i++)
        hashtable[i] =  - 1;
                                                       /*哈希表中初始位置全置-1*/
    for (i = 0; i < 50; i++)
        flag[i] = 0;
                                                       /*50 以内所有数未产生时均标志为 0*/
    srand((unsigned long)time(0));                     /*利用系统时间做种子产生随机数*/
    i = 0;
    while (i != N)
    {
        t = rand() % 50;                               /*产生一个 50 以内的随机数赋给 t*/
        if (flag[t] == 0)
                                                       /*查看 t 是否产生过*/
        {
            creathash(t);                              /*调用函数创建哈希表*/
```

```
            printf("%2d:", t);                          /*将该元素输出*/
            for (j = 0; j < Max; j++)
                printf("(%2d) ", hashtable[j]);
                                                        /*输出哈希表中内容*/
            printf("\n");
            flag[t] = 1;                                /*将产生的这个数标志为 1*/
            i++;                                        /*i 自加*/
        }
    }
    printf("please input number which do you want to search:");
    scanf("%d", &t);                                    /*输入要查找的元素*/
    if (t > 0 && t < 50)
    {
        i = search(t);                                  /*调用 search 进行哈希查找*/
        if (i != - 1)
            printf("success!The position is:%d\n", i);
                                                        /*若查找到该元素则输出其位置*/
        else
            printf("sorry,no found!");
                                                        /*未找到输出提示信息*/
    }
    else
        printf("inpput error!");
}
```

举一反三

根据本实例，读者可以：

- 采用哈希查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 146 的元素；
- 采用哈希查找法在关键字序列 45、85、63、785、42、654、219、146、325、795 中查找关键字为 63 的元素。

算法 11　斐波那契数列

实例说明

斐波那契数列的特点是第 1、2 两个数为 1、1，从第 3 个数开始，该数是前两个数之和。求这个数列的前 30 个元素。运行结果如图 15.11 所示。

```
C:\ TC
     1      1      2      3      5
     8     13     21     34     55
    89    144    233    377    610
   987   1597   2584   4181   6765
 10946  17711  28657  46368  75025
121393 196418 317811 514229 832040
```

图 15.11　斐波那契数列

技术要点

分析题目中的要求，我们可以用如下等式来表示斐波那契数列。

$$F_1=1 \qquad (n=1)$$

$$F_2=1 \qquad (n=2)$$

$$F_n=F_{n-1}+F_{n-2} \qquad (n\geq 3)$$

这里面我们将 F 的下标看成数组的下标即可完成该程序。

实现过程

例 15.11 斐波那契数列算法（光盘位置：光盘\MR\Instance\15\11）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）程序中用到两个 for 循环语句，第一个 for 循环是实现从第 3 项开始每一项等于前两项之和，第二个 for 循环是将存储在数组中的数据以 5 个一行的形式输出。

（4）主要程序代码如下。

```
main()
{
    int i;                                          /*定义整型变量 i*/
    long f[31];                                     /*意义数组为长整型*/
    f[1] = 1, f[2] = 1;                             /*数组中的 f[1]、f[2]赋初值为 1*/
    for (i = 3; i < 31; i++)
        f[i] = f[i - 1] + f[i - 2];                 /*数列中从第 3 项开始每一项等于前两项之和*/
    for (i = 1; i < 31; i++)
    {
        printf("%10ld", f[i]);                      /*输出数组中的 30 个元素*/
        if (i % 5 == 0)
            printf("\n");                           /*每 5 个元素进行一次换行*/
    }
}
```

举一反三

根据本实例，读者可以：

- 从键盘中任意输入两个数，输出包括这两个数在内的 20 个数据，这 20 个数据有如下关

系：输入的两个数不变，第三个数等于第二个数减第一个数，第四个数等于第三个数减去第二个数，依次类推；

- 从键盘中任意输入两个数，输出包括这两个数在内的 10 个数据，这 10 个数据有如下关系：输入的两个数不变，第三个数等于第二个数与第一个数的乘积再除以 4，第四个数等于第三个数与第二个数的乘积再除以 4，依次类推。

算法 12　角谷猜想

实例说明

角谷猜想的内容是：任给一个自然数，若为偶数则除以 2，若为奇数则乘 3 加 1，得到一个新的自然数后按照上面的法则继续演算，若干次后得到的结果必然为 1。编程验证该定理。运行结果如图 15.12 所示。

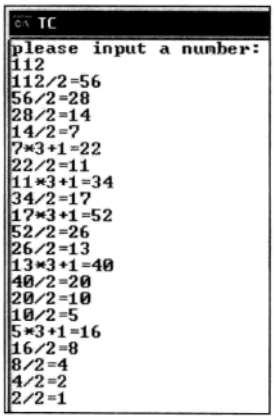


图 15.12　角谷猜想

技术要点

本实例没有太多难点，只需根据题中给的条件编写程序即可，这里只强调一点就是判断一个数是奇数还是偶数，程序中采用的是对 2 取余的方法，若余数为 0 则说明该数是偶数，否则该数为奇数。

实现过程

例 15.12　角谷猜想（光盘位置：光盘\MR\Instance\15\12）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）程序中采用 while 循环来判断每次运算所得到的结果是否为 1，当不为 1 时继续按照题中给的条件进行判断，直到最终结果等于 1。

（4）主要程序代码如下。

```
main()
{
    long i, n;                                  /*定义变量为长整型*/
    printf("please input a number:\n");
    scanf("%ld", &n);                           /*从键盘中任意输入一长整型数*/
    while (n != 1)                              /*当最终结果不为 1 时一直执行循环体语句*/
```

```
    {
        if (n % 2 == 0)                                    /*判断 n 是否为偶数*/
        {
            printf("%ld/2=%ld\n", n, n / 2);
            n = n / 2;                                     /*当 n 为偶数时 n 除以 2*/
        }
        else
        {
            printf("%ld*3+1=%ld\n", n, n *3+1);
            n = n * 3+1;                                   /*当 n 为奇数时乘以 3 加 1*/
        }
    }
}
```

举一反三

根据本实例，读者可以：

- 求 13 的 13 次方的最后三位；
- 判断一个 5 位数是不是回文数。

算法 13 歌德巴赫猜想

实例说明

验证 100 以内的正偶数都能分解为两个素数之和，即验证歌德巴赫猜想对 100 以内的正偶数成立。运行结果的后 5 行如图 15.13 所示。

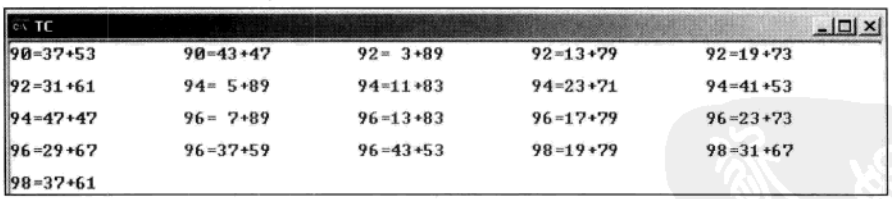


图 15.13 歌德巴赫猜想

技术要点

为了验证歌德巴赫猜想对 100 以内的正偶数是成立的，要将正偶数分解为两部分，再对这两部分进行判断，如果均是素数则满足题意，若不是则重新分解继续判断。本实例把素数的判断过程自定义到函数 ss()中，对每次分解出的两个数只要调用函数 ss()来判断即可。

实现过程

例 15.13　哥德巴赫猜想（光盘位置：光盘\MR\Instance\15\13）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>
```

（3）自定义 ss()函数，函数类型为基本整型，作用是判断一个数是否为素数。代码如下。

```
int ss(int i)                                          /*自定义函数判断是否为素数*/
{
    int j;
    if (i <= 1)                                        /*小于 1 的数不是素数*/
        return 0;
    if (i == 2)                                        /*2 是素数*/
        return 1;
    for (j = 2; j < i; j++)                            /*对大于 2 的数进行判断*/
    {
        if (i % j == 0)
            return 0;
        else if (i != j + 1)
            continue;
        else
            return 1;
    }
}
```

（4）对 6～100 之间的正偶数进行拆分，再对拆分出来的数分别调用函数 ss()进行是否为素数的判断，若均为素数，则按指定格式输出，否则继续下次循环，重新进行拆分及判断。

（5）主要程序代码如下。

```
main()
{
    int i, j, k, flag1, flag2, n = 0;
    for (i = 6; i < 100; i += 2)
    for (k = 2; k <= i / 2; k++)
    {
        j = i - k;
        flag1 = ss(k);                                 /*判断拆分出的数是否是素数*/
        if (flag1)
        {
            flag2 = ss(j);
            if (flag2)                                 /*如果拆分出的两个数均是素数则输出*/
            {
                printf("%3d=%3d+%3d,", i, k, j);
                n++;
                if (n % 5 == 0)
                    printf("\n");
            }
        }
    }
}
```

举一反三

根据本实例，读者可以：

- 取一个整数 a 从右端开始的 4～7 位；
- 有 n 个整数，使其前面各数顺序向后移 m 个位置，最后 m 个数变成最前面的 m 个数。

算法 14 四方定理

实例说明

四方定理的内容是：所有的自然数至多只要用 4 个数的平方和就可以表示。编程验证该定理。运行结果如图 15.14 所示。

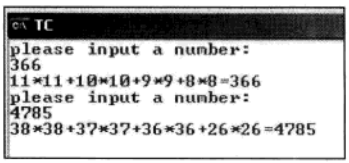


图 15.14 四方定理

技术要点

本实例对 4 个变量 i、j、k、l 采用穷举试探的方法进行计算，当满足定理中的条件时输出计算结果。

实现过程

例 15.14 四方定理（光盘位置：光盘\MR\Instance\15\14）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）程序中采用 4 个 for 循环语句分别对 i、j、k、l 的值进行试探，对满足 if 条件语句中条件的将其按指定格式输出，输出完退出程序。

（4）主要程序代码如下。

```
main()
{
    long i, j, k, l, n;                                    /*定义变量为长整型*/
    printf("please input a number:\n");
    scanf("%ld", &n);                                      /*定义变量为长整型*/
    for (i = 0; i <= n; i++)                               /*对 i, j, k, l 进行穷举*/
```

```
            for (j = 0; j <= i; j++)
                for (k = 0; k <= j; k++)
                    for (l = 0; l <= k; l++)
                        if (i *i + j * j + k * k + l * l == n)       /*判断是否满足定理要求 */
        {
            /*将满足要求的结果输出*/
            printf("%ld*%ld+%ld*%ld+%ld*%ld+%ld*%ld=%ld\n", i, i, j, j, k, k, l, l,n);
            exit(0);
        }
    }
```

举一反三

根据本实例，读者可以：

- 编程实现输入 n 为偶数时，调用函数求 1/2+1/4+…+1/n，当输入 n 为奇数时，调用函数 1/1+1/3+…+1/n；
- 编程计算 1000！末尾有多少 0。

算法 15　尼科彻斯定理

实例说明

尼科彻斯定理内容是：任何一个整数的立方都可以写成一串连续奇数的和。编程验证该定理。运行结果如图 15.15 所示。

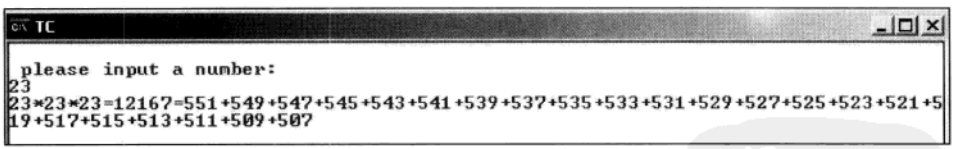


图 15.15　尼科彻斯定理

技术要点

解决本实例的关键是先要确定这串连续奇数中的最大值的范围，这里我们可以这样分析：任何立方值（这里设为 *sum*）的一半（这里设为 x）如果是奇数，则 $x+x+2$ 的值一定大于 *sum*，那么这串连续奇数的最大值不会超过 x；如果 x 是偶数，我们需把它变成奇数，那么变成奇数到底是加 1、减 1 还是其他呢？这里我们选择加 1，因为 $x+1+x-1$ 正好等于 *sum*，所以当 x 是偶数时这串连续奇数的最大值不会超过 $x+1$。在确定了范围后我们就可以从最大值开始进行穷举。

实现过程

例 15.15 尼科彻斯定理（光盘位置：光盘\MR\Instance\15\15）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）程序中使用了两个 while 循环语句，第一个 while 语句的功能是从可能的最大值开始进行穷举，直到 1 为止，第二个 while 循环的功能是通过第一个 while 循环所确定的 *i* 值进行每次减 2 逐次累加求和，当累加和等于立方值时输出累加过程并跳出循环，当累加和大于立方值时也同样跳出第二个循环回到第一个循环中。

（4）主要程序代码如下。

```
main()
{
    int i, j, k = 0, l, n, m, sum,flag=1;
    printf("\n please input a number:\n");
    scanf("%d", &n);                                    /*从键盘中任意输入一个数*/
    m = n * n * n;                                      /*计算出该数的立方*/
    i = m / 2;
    if (i % 2 == 0)                                     /*当 i 为偶数时 i 值加 1*/
        i = i + 1;
    while (flag==1&&i >= 1)                             /*当 i 大于等于 1 且 flag=1 时执行循环体语句*/
    {
        sum = 0;
        k = 0;
        while (1)
        {
            sum += (i - 2 * k);                         /*奇数累加求和*/
            k++;
            if (sum == m)                               /*如果 sum 与 m 相等，则输出累加过程*/
            {
                printf("%d*%d*%d=%d=", n, n, n, m);
                for (l = 0; l < k - 1; l++)
                    printf("%d+", i - l * 2);
                printf("%d\n", i - (k - 1) *2);         /*输出累加求和的最后一个数*/
                flag=0;
                break;
            }
            if (sum > m)
                break;
        }
        i -= 2;                                         /*i 等于下一个奇数继续上面过程*/
    }
}
```

举一反三

根据本实例，读者可以：

- 输入任意数 n，求 $n!$ 末尾有多少 0；
- 一个 4 位数前两位数相同，后两位数相同，相互间又不同，且是另一个整数的平方，求该 4 位数。

算法 16　创建单向链表

实例说明

本实例实现创建一个简单的链表，并将这个链表中数据输出到窗体上。程序运行效果如图 15.16 所示。

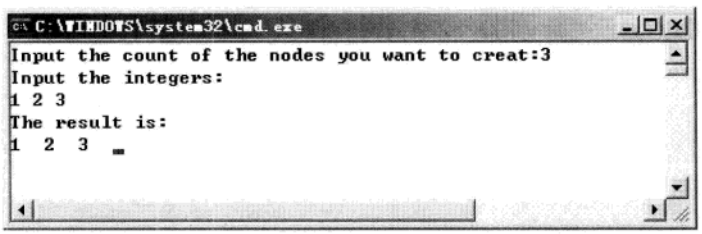


图 15.16　创建并显示链表

技术要点

链表是动态分配存储空间的链式存储结构，其包括一个“头指针”变量，头指针中存放一个地址，该地址指向一个元素。链表中每一个元素称为“节点”，每个节点都由两部分组成：存储数据元素的数据域和直接后继存储位置的指针域。指针域中存储的即是链表的下一个节点的存储位置，是一个指针。多个节点结成一个链表。链表的结构示意图如图 15.17 所示。



图 15.17　链表结构示意图

其中第 0 个节点称为整个链表的头节点，它一般不存放具体数据，只是存放第一个节点的地址，也称为“头指针”。最后一个节点的指针域设置为空（NULL），作为链表的结束标志，表示它没有后继节点。

说明：从图 15.17 可以看出，链表并不一定是连续存放的存储节点，并且只要获得链表的头节点，就可以通过指针定位整个链表。

使用结构体变量作为链表中的节点，因为结构体变量成员可以是数值类型、字符类型、数组类

型，也可以是指针类型，这样就可以使用指针类型成员来存放下一个节点的地址，使用其他类型成员存放数据信息。例如，一个结构体类型可以如下。

```
struct LNode
{
    int data;
    struct LNode *next;
};
```

上面的结构体类型中，成员 data 用来存放节点中的数据，相当于图 15.17 中的 *a*、*b*、*c*。next 是指针类型的成员，它指向 struct LNode 类型数据，就是本结构体类型的数据。

注意：上面只是定义了一个结构体类型，并未实际分配存储空间，只有定义了变量后才分配存储空间。

在创建链表时要动态地为链表分配空间，C 语言的库函数提供了几种函数来实现动态开辟存储单元。这里使用 malloc()函数实现动态地开辟存储单元，下面进行介绍。

malloc 函数原型如下所示。

```
void *malloc(unsigned int size);
```

其作用是在内存的动态存储区中分配一个长度为 size 的连续空间。函数返回值是一个指向分配域起始地址的指针（类型为 void）。如果分配空间失败（例如，内存空间不足），则返回空指针（NULL）。

实现过程

例 15.16 创建单向链表（光盘位置：光盘\MR\Instance\15\16）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include "stdio.h"
#include <malloc.h>
```

（3）声明 struct LNode 类型。

```
struct LNode
{
    int data;
    struct LNode *next;
};
```

（4）创建 create()自定义函数，用于创建一个链表，将此函数定义为指针类型，使其返回值为

指针类型，返回值指向一个 struct LNode 类型数据，实际上是返回链表的头指针。

```
struct LNode *create(int n)
{
    int i;
    struct LNode *head,  *p1,  *p2;
    int a;
    head = NULL;
    printf("Input the integers:\n");
    for (i = n; i > 0; --i)
    {
        p1 = (struct LNode*)malloc(sizeof(struct LNode));        /*分配空间*/
        scanf("%d", &a);                                         /*输入数据*/
        p1->data = a;                                            /*数据域赋值*/
        if (head == NULL)                                        /*指定头节点*/
        {
            head = p1;
            p2 = p1;
        }
        else
        {
            p2->next = p1;                                       /*指定后继指针*/
            p2 = p1;
        }
    }
    p2->next = NULL;
    return head;
}
```

（4）创建 main()函数作为程序的入口程序，在 main()函数中调用 create()自定义函数，实现创建一个链表，并将链表中的数据输出。

实现代码如下。

```
void main()
{
    int n;
    struct LNode *q;
    printf("Input the count of the nodes you want to creat:");
    scanf("%d", &n);                                             /*输入链表节点个数*/
    q = create(n);
    printf("The result is:\n");
    while (q)
    {
        printf("%d  ", q->data);                                 /*输出链表*/
        q = q->next;
    } getch();
}
```

举一反三

根据本实例，读者可以：

- 使用学生信息动态创建一个单链表；
- 创建固定长度单链表。

算法 17 约瑟夫环

实例说明

本实例使用循环链表实现约瑟夫环。给定的一组编号分别是 4、7、5、9、3、2、6、1、8。报数初始值由用户输入，这里输入 4，如图 15.18 所示。按照约瑟夫环原理打印输出队列。

程序运行结果如图 15.18 所示。

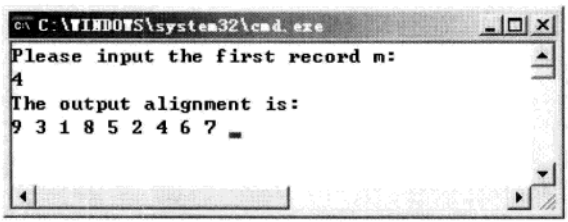


图 15.18 按照约瑟夫环原理打印输出队列

技术要点

约瑟夫环算法是：n 个人围成一圈，每个人都有一个互不相同的密码，该密码是一个整数值，选择一个人作为起点，然后顺时针从 1 到 k（k 为起点人手中的密码值）数数。数到 k 的人退出圈子，然后从下一个人开始继续从 1 到 j（刚退出圈子的人的密码）数数，数到 j 的人退出圈子。重复上面的过程。直到剩下最后一个人。

实现过程

例 15.17 约瑟夫环（光盘位置：光盘\MR\Instance\15\17）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，声明结构体，声明自定义函数。

```
#include "stdio.h"
#define N 9
#define OVERFLOW 0
#define OK 1
int KeyW[N]={4,7,5,9,3,2,6,1,8};
void Joseph(LinkList p,int m, int x);
```

（3）声明 struct LNode 类型。

```
typedef struct LNode{
    int keyword;
```

```
    struct LNode *next;
}LNode,*LinkList;
```

（4）创建 Joseph()自定义函数，使用循环链表实现约瑟夫环算法，根据给定数获得一个数列。

```
void Joseph(LinkList p,int m,int x){
    LinkList q;                                    /*声明变量*/
    int i;
    if(x==0)return;
    q=p;
    m%=x;
    if(m==0)m=x;
    for(i=1;i<=m;i++){                             /*找到下一个节点*/
        p=q;
        q=p->next;
    }
    p->next=q->next;
    i=q->keyword;
    printf("%d ",q->keyword);
    free(q);
    Joseph(p,i,x-1);                               /*递归调用*/
}
```

（5）创建 main()函数作为程序的入口函数，调用 Joseph()自定义函数实现约瑟夫环算法，并将得到的数列输出。

```
int main()
{
    int i,m;
    LinkList Lhead,p,q;
    Lhead=(LinkList)malloc(sizeof(LNode));                    /*申请节点空间*/
    if(!Lhead) return OVERFLOW;
    Lhead->keyword=KeyW[0];                                   /*数据域赋值*/
    Lhead->next=NULL;
    p=Lhead;
    for(i=1;i<9;i++){                                         /*创建循环链表*/
        if(!(q=(LinkList)malloc(sizeof(LNode))))return OVERFLOW;
        q->keyword=KeyW[i];
        p->next=q;
        p=q;
    }
    p->next=Lhead;
    printf("Please input the first record m: \n");
    scanf("%d",&m);
    printf("The output alignment is:\n");
    Joseph(p,m,N);
    getch();
    return OK;
}
```

举一反三

根据本实例，读者可以：

- 循环链表的合并；

- 查找循环链表中的元素值。

算法 18 双链表逆序输出

实例说明

本实例实现创建一个指定节点数的双向链表，并将双向链表中的节点数据逆序输出。程序运行结果如图 15.19 所示。

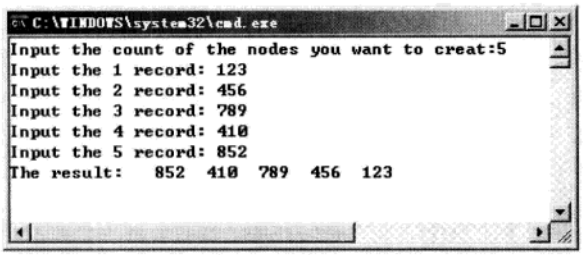


图 15.19 双链表逆序输出

技术要点

双向链表节点数据逆序输出的实现算法很简单，因为双向链表有两条指针链，一条可以看成是从头节点出发直到尾节点的指针链，另一条可以看成是从尾节点出发直到指向头节点的指针链。双向链表节点数据的逆序输出就是根据双向链表的这个特性实现的，从尾节点到头节点使用指针依次遍历该链表的节点，即可实现逆序查找双向链表节点，并将节点数据域的数据信息输出。

实现过程

例 15.18 双向链表逆序输出（光盘位置：光盘\MR\Instance\15\18）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件，定义宏。

```
#include <stdio.h>
#define N 10
```

（3）声明双向链表结构。

```
typedef struct node
{
```

```
    char name[20];
    struct node *prior,  *next;
} stud;                                          /*双向链表的结构定义*/
```

（4）创建 create()自定义函数，创建一个双向链表，其返回值为指针类型，返回值指向一个 struct node 类型数据，实际上是返回链表的头指针。

```
/*双向链表的创建*/
stud *creat(int n)
{
    stud *p,  *h,  *s;                           /*声明双向链表结构类型的指针*/
    int i;
    h = (stud*)malloc(sizeof(stud));             /*创建头节点*/
    /*初始化头节点*/
    h->name[0] = '\0';
    h->prior = NULL;
    h->next = NULL;
    p = h;                                       /*p 指向头节点*/
    printf("Input the records:\n ");
    for (i = 0; i < n; i++)
    {
        s = (stud*)malloc(sizeof(stud));         /*申请节点空间*/
        p->next = s;
        scanf("%s", s->name);                    /*输入数据*/
        s->prior = p;                            /*指定前驱节点*/
        s->next = NULL;                          /*指定后继节点*/
        p = s;
    }
    p->next = NULL;
    return (h);                                  /*返回头节点*/
}
```

（5）创建自定义函数 gettp()，实现获取一个指向双向链表尾节点的指针。程序代码如下。

```
stud *gettp(stud *head)
{
    stud *p,  *r;
    while (p->next != NULL)
    {
        p = p->next;
    }
    return p;                                    /*返回尾节点指针*/
}
```

（6）创建 main()函数，作为程序的入口函数。在主函数中调用 creat()自定义函数和 gettp()自定义函数，实现创建并找到双向链表的尾节点。使用双向链表节点中的指向直接前驱的指针遍历双向链表，并将节点的数据依次输出到窗体上。

```
void main()
{
    int n, i;
    int x;
    stud *q;
    printf("Input the count of the nodes you want to creat:");
    scanf("%d", &n);                             /*输入要创建链表的节点数*/
    q = creat(n);                                /*创建双向链表*/
    q = gettp(q);                                /*找到双向链表的尾节点*/
    printf("The result: ");
    while (q)
```

```
    {
        printf("  %s", &*(q->name));          /*逆序输出*/
        q = q->prior;                          /*从尾节点开始向前遍历链表节点*/
    }
    getch();
}
```

举一反三

根据本实例，读者可以：

- 获取双向链表节点个数；
- 判断双向链表是否为空。

算法 19 二叉树的遍历

实例说明

构造一棵二叉树，分别采用先序遍历、中序遍历和后序遍历来遍历该二叉树。运行结果如图 15.20 所示。

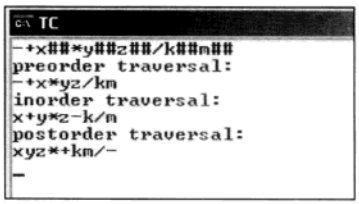


图 15.20 二叉树遍历

技术要点

在遍历二叉树时若先左后右，则有 3 种遍历方法，分别为先序遍历、中序遍历和后序遍历，它们的递归定义如下。

- 先序遍历：若二叉树非空，则先访问根节点，再按先序遍历左子树，最后按前序遍历右子树。
- 中序遍历：若二叉树非空，则先按中序遍历左子树，再访问根节点，最后按中序遍历右子树。
- 后序遍历：若二叉树非空，则先按后序遍历左子树，再按后序遍历右子树，最后访问根节点。

实现过程

例 15.19 二叉树的遍历（光盘位置：光盘\MR\Instance\15\19）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
#include <stdlib.h>
```

（3）二叉链表的结构声明，代码如下。

```
typedef struct node                                    /*二叉链表结构声明*/
{
    struct node *lchild;
    char data;
    struct node *rchild;
} bitnode,  *bitree;                                   /*bitnode、bitree 为该结构体类型*/
```

（4）自定义 CreatTree()函数，用于构造二叉树，代码如下。

```
bitree CreatTree()                                     /*构造二叉树*/
{
    char a;
    bitree new;
    scanf("%c", &a);
    if (a == '#')
        return NULL;
    else
    {
        new = (bitree)malloc(sizeof(bitnode));
        new->data = a;
        new->lchild = CreatTree();                     /*递归创建左子树*/
        new->rchild = CreatTree();                     /*递归创建右子树*/
    }
    return new;
}
```

（5）自定义 preorderTraverse()函数，用于先序遍历二叉树，代码如下。

```
void preorderTraverse(bitree bt)                       /*先序遍历二叉树*/
{
    if (bt != NULL)
    {
        printf("%c", bt->data);                        /*访问根节点*/
        preorderTraverse(bt->lchild);                  /*访问左子树*/
        preorderTraverse(bt->rchild);                  /*访问右子树*/
    }
}
```

（6）自定义 inorderTraverse()函数，用于中序遍历二叉树，代码如下。

```
void InorderTraverse(bitree bt)                        /*中序遍历二叉树*/
{
    if (bt != NULL)
    {
        InorderTraverse(bt->lchild);                   /*访问左子树*/
        printf("%c", bt->data);                        /*访问根节点*/
        InorderTraverse(bt->rchild);                   /*访问右子树*/
    }
}
```

（7）自定义 postorderTraverse()函数，用于后序遍历二叉树，代码如下。

```
void postorderTraverse(bitree bt)
{
    if (bt != NULL)
    {
        postorderTraverse(bt->lchild);          /*访问左子树*/
        postorderTraverse(bt->rchild);          /*访问右子树*/
        printf("%c", bt->data);                 /*访问根节点*/
    }
}
```

（8）主函数程序代码如下。

```
main()
{
    bitree root;
    root = CreatTree();                         /*调用 CreatTree 函数构造二叉树*/
    printf("preorder traversal:\n");
    preorderTraverse(root);                     /*调用 preorderTraverse 函数先序遍历二叉树*/
    printf("\n");
    printf("inorder traversal:\n");
    InorderTraverse(root);                      /*调用 inorderTraverse 函数中序遍历二叉树*/
    printf("\n");
    printf("postorder traversal:\n");
    postorderTraverse(root);                    /*调用 postorderTraverse 函数后序遍历二叉树*/
    printf("\n");
}
```

举一反三

根据本实例，读者可以：

- 已知一棵二叉树的中序遍历和后序遍历分别为 BDCEAFHG 和 DECBHGFA，编程求出该二叉树前序遍历的结果；
- 编程求如图 15.21 所示的二叉树的前序、中序及后序遍历的结果。

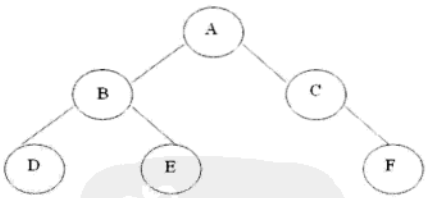


图 15.21　二叉树

算法 20　用栈设置密码

实例说明

本实例使用栈设置一个密码，当输入错误密码时，系统提示密码错误，输入错误 3 次退出。输入了正确的密码后，显示密码正确。程序密码为“13579”，程序运行结果如图 15.22 所示。

技术要点

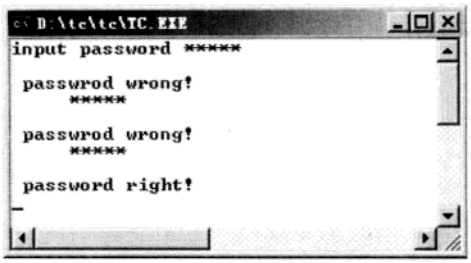


图 15.22　设置密码程序示意图

本实例使用栈来设置密码，应用到了栈的定义、初始化、进栈、出栈等功能。这里将这些功能设置成了单个的自定义函数，并在相应的位置进行调用。本实例首先定义一个密码字符串，将键盘上输入的密码压倒栈中，将栈中数据与密码字符串进行比较，看密码是否正确。输入错误 3 次则退出。

实现过程

例 15.20　用栈设置密码（光盘位置：光盘\MR\Instance\15\20）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
#include<string.h>
#include<conio.h>
#include<stdlib.h>
```

（3）预定义全局变量。

```
#define STACK_SIZE 100                    /*假定预分配的栈空间最多为 100 个元素*/
char PASSWORD[10]= "13579";
typedef char DataType;                    /*假定栈元素的数据类型为字符*/
```

（4）声明 struct student 类型。

```
typedef struct{
      DataType *base;
      DataType *top;
      int stacksize;
      int length;
}SeqStack;
```

（5）创建自定义函数实现构造栈、进栈、出栈、取栈顶元素等操作。实现代码如下。

```
void Initial(SeqStack *s)
{
      s->base=(DataType *)malloc(STACK_SIZE * sizeof(DataType));
      if(!s->base) exit (-1);
      s->top=s->base;
      s->stacksize=STACK_SIZE;
      s->length=0;

}

/*判栈空*/
```

```
int IsEmpty(SeqStack *S)
{
    return S->top==S->base;
}

/*判栈满*/
int IsFull(SeqStack *S)
{
    return S->top-S->base==STACK_SIZE-1;
}

/*进栈*/
void Push(SeqStack *S,DataType x)
{
    if (IsFull(S))
    {
        printf("栈上溢");                    /*上溢,退出运行*/
        exit(1);
    }
  *(S->top++) =x;                            /*栈顶指针加 1 后将 x 入栈*/
  ++S->length;
  /* printf("%c",*S->top);*/
}

/*出栈*/
DataType Pop(SeqStack *S)
{
    if(IsEmpty(S))
    {
        printf("栈为空");                    /*下溢,退出运行*/
        exit(1);
    }
    --(S->length);

    return *--S->top;                        /*栈顶元素返回后将栈顶指针减 1*/
}

/* 取栈顶元素*/
DataType GetTop(SeqStack *S,DataType *e)
{
    if(IsEmpty(S))
    {
        printf("栈为空");                    /*下溢,退出运行*/
        exit(1);
    }
    *e= *(S->top-1);
    S->top--;
}
void change(SeqStack *s,char *a)
{
    int n=s->length-1;
    while(!IsEmpty(s))
    {
    GetTop(s,&a[n--]);}                      /*获取栈顶元素*/


}
void clearstack(SeqStack *s)
{
    s->top=s->base;                          /*清空栈*/
    s->length=0;                             /*设置栈的长度为 0*/

}
```

创建 PwdSet()自定义函数，实现使用栈判断用户输入的密码是否正确，程序代码如下。

```
void PwdSet(SeqStack *s)
{
    int i=0,k,j=0;
    DataType ch,*a;
    k=strlen(PASSWORD);
    printf("input password ");
    for(;;)
    {
    if(i>=3)
    {
         i++;
         break;
         }
    else if(i>0 && i<3)
    {

    for(j=1;j<=s->length;j++)printf(" ");
    clearstack(s);                              /*清空栈*/
    }
    for(;;)
    {
         ch=getch();
         if(ch!=13)
         {
             if(ch==8){
                   Pop(s);
                   gotoxy(4+j,2);
                   printf(" ");
                   gotoxy(4+j,2);
             }
             else{printf("*");Push(s,ch);}
             j=s->length;

         }
         else
         {printf("\n");
         break;}
    }
    i++;
    if(k!=j)continue;
    else
    {
         a=(DataType *)malloc(s->length * sizeof(DataType));
         change(s,a);

         for(j=1;j<=s->length;)
         {

             if(*(a+j-1)==PASSWORD[j-1]) j++;
             else
             {

                   j=s->length+2;
                   printf("\n passwrod wrong!\n");
                   break;
             }
         }
         if(j==s->length+2)continue;
```

```
            else break;
        }
        }
        if(i==4)printf("\n Have no chance,It will quit!");
        else printf("\n password right!\n");
        free(a);
    }
```

（6）创建 main()函数作为程序的入口函数，调用 PwdSet()自定义函数来判断用户输入的密码是否正确。程序代码如下。

```
main()
{
    LinkList L1;

    L1 = create();
    printf("The linklist is:\n");
    while (L1)
    {
        printf("%c ", L1->num);
        L1 = L1->next;
    }
    getch();
}
```

举一反三

根据本实例，读者可以：

- 用数组仿真堆栈；
- 动态堆栈。

算法 21 链队列

实例说明

采用链式存储法编程实现元素入队、出队以及将队列中的元素显示出来，要求整个过程以菜单选择的形式实现。运行结果如图 15.23、图 15.24、图 15.25、图 15.26 所示。

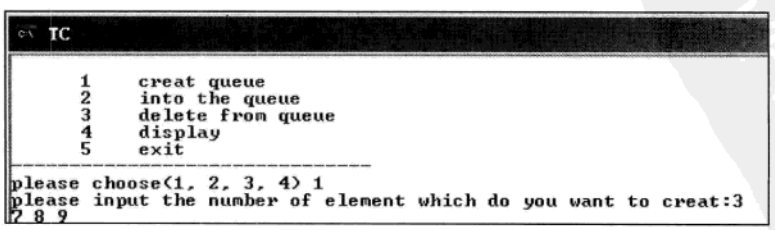


图 15.23 创建队列

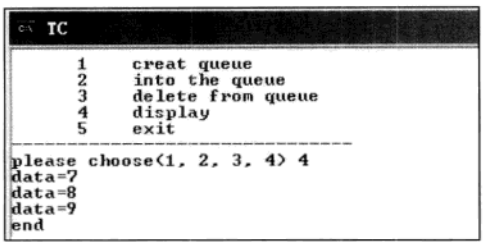


图 15.24　显示队列中元素

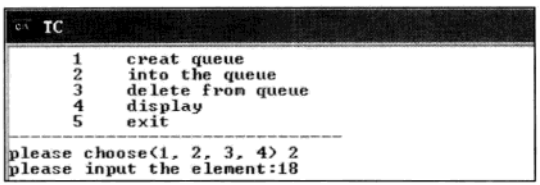


图 15.25　元素入队列

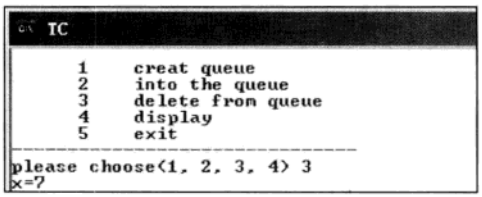


图 15.26　删除队列中元素

技术要点

队列的链式存储结构是通过由节点构成的单链表实现的，此时只允许在单链表的表首进行删除，在单链表的表尾进行插入，因此需要使用两个指针：队首指针 front 和队尾指针 rear。用 front 指向队首节点的存储位置，用 rear 指向队尾节点的存储位置。

实现过程

例 15.21　链队列（光盘位置：光盘\MR\Instance\15\21）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
#include <stdlib.h>
```

（3）定义结构体 node 及 quefr，分别存储入队元素内容及队首队尾指针，代码如下。

```
typedef struct node                          /*定义节点*/
{
    ElemType data;                           /*存放元素内容*/
    struct node *next;                       /*指向下个节点*/
} quenode;

struct quefr                                 /*定义节点存放队首队尾指针*/
```

```
{
    quenode *front,  *rear;
};
```

（4）自定义 creat()函数，作用是初始化链队列，代码如下。

```
void creat(struct quefr *q)                    /*自定义函数初始化链队列*/
{
    quenode *h;
    h = (quenode*)malloc(sizeof(quenode));
    h->next = NULL;
    q->front = h;                              /*队首指针队尾指针均指向头节点*/
    q->rear = h;
}
```

（5）自定义 enqueue()函数，作用是元素入队列，代码如下。

```
void enqueue(struct quefr *q, int x)           /*自定义函数，元素 x 入队*/
{
    quenode *s;
    s = (quenode*)malloc(sizeof(quenode));
    s->data = x;                               /*x 放到节点的数据域中*/
    s->next = NULL;                            /*next 域为空*/
    q->rear->next = s;
    q->rear = s;                               /*队尾指向 s 节点*/
}
```

（6）自定义 dequeue()函数，作用是元素出队列，代码如下。

```
ElemType dequeue(struct quefr *q)              /*自定义函数实现元素出队*/
{
    quenode *p;
    ElemType x;
    p = (quenode*)malloc(sizeof(quenode));
    if (q->front == q->rear)
    {
        printf("queue is NULL \n");
        x = 0;
    }
    else
    {
        p = q->front->next;
        q->front->next = p->next;              /*指向出队元素所在节点的后一个节点*/
        if (p->next == NULL)
            q->rear = q->front;
        x = p->data;
        free(p); /*释放 p 节点*/
    }
    return (x);
}
```

（7）自定义 display()函数，作用是显示队列中的元素，代码如下。

```
void display(struct quefr dq)
 /*自定义函数显示队列中元素*/
{
    quenode *p;
    p = (quenode*)malloc(sizeof(quenode));
    p = dq.front->next;                        /*指向第一个数据元素节点 */
    while (p != NULL)
```

```
    {
        printf("data=%d\n", p->data);
        p = p->next;                                    /*指向下个节点*/
    } printf("end \n");
}
```

（8）主函数程序代码如下。

```
main()
{
    struct quefr *que;
    int n, i, x, sel;
    void display();                                     /*显示队列中元素*/
    void creat();                                       /*创建队列*/
    void enqueue();                                     /*元素入队列*/
    ElemType dequeue();                                 /*元素出队列*/
    do
    {
        printf("\n");
        printf("      1    creat queue     \n");
        printf("      2    into the queue     \n");
        printf("      3    delete from queue  \n");
        printf("      4    display\n");
        printf("      5    exit     \n");
        printf("-----------------------------\n");
        printf("please choose(1, 2, 3, 4, 5) ");
        scanf("%d", &sel);                              /*输入相关功能所对应的标号*/
        switch (sel)
        {
            case 1:
                que = (struct quefr*)malloc(sizeof(struct quefr));
                /*分配内存空间*/
                creat(que);                             /*初始化队列*/
                printf(
                    "please input the number of element which do you want to creat:");
                scanf("%d", &n);                        /*输入队列元素个数*/
                for (i = 1; i <= n; i++)
                {
                    scanf("%d", &x);                    /*输入元素*/
                    enqueue(que, x);
                }
                break;
            case 2:
                printf("please input the element:");
                scanf("%d", &x);                        /*输入元素*/
                enqueue(que, x);                        /*元素入队*/
                break;
            case 3:
                printf("x=%d\n", dequeue(que));         /*元素出队*/
                break;
            case 4:
                display(*que);                          /*显示队列中元素*/
                break;
            case 5:
                exit(0);
        }
    }
    while (sel <= 4);
}
```

举一反三

根据本实例，读者可以：

- 利用队列，编程实现模拟飞机售票；
- 编写顺序队列入队出队过程。

算法 22 SHA 加密

实例说明

安全哈希算法（Secure Hash Algorithm）主要适用于数字签名标准（Digital Signature Standard，DSS）里面定义的数字签名算法（Digital Signature Algorithm，DSA）。对于长度小于 2^{64} 位的消息，SHA1 会产生一个 160 位的消息摘要。当接收到消息的时候，这个消息摘要可以用来验证数据的完整性。在传输的过程中，数据很可能会发生变化，那么这时候就会产生不同的消息摘要。运行本实例编译后的可执行文件，执行参数为“C:\11.txt”，将文件 11.txt 进行 SHA1 加密，文件 11.txt 中内容如图 15.27 所示，程序运行结果如图 15.28 所示。

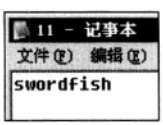


图 15.27 文件 11.txt 内容

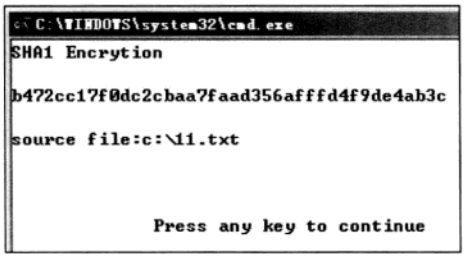


图 15.28 SHA1 加密效果

技术要点

在 SHA1 算法中，必须把原始消息（字符串、文件等）转换成位字符串。假设对字符串“abc”产生消息摘要。首先，将它转换成位字符串：01100001 01100010 01100011，它们用十进制表示分别为 97、98、99，分别是 a、b、c 的 ASCII 码。

位字符串的长度为 24。下面需要 5 个步骤来计算。

（1）补位

消息必须进行补位，以使其长度在对 512 取模以后的余数是 448。也就是说，补位后的消息长

度%512 = 448。即使长度已经满足对 512 取模后余数是 448，补位也必须要进行。

补位是这样进行的：先补一个 1，然后再补 0，直到长度满足对 512 取模后余数是 448 为止。总而言之，补位是至少补一位，最多补 512 位。还是以前面的“abc”为例显示补位的过程。

原始信息：01100001 01100010 01100011

补位第一步：01100001 01100010 01100011 1

首先补一个“1”

补位第二步：01100001 01100010 01100011 10……0

然后补 423 个“0”

可以把最后补位完成后的数据用十六进制写成下面的样子。

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000

现在，数据的长度是 448 了，可以进行下一步操作。

（2）补长度

所谓的补长度是将原始数据的长度补到已经进行了补位操作的消息后面。通常用一个 64 位的数据来表示原始消息的长度。如果消息长度不大于 2^{64}，那么第一个字就是 0。在进行了补长度的操作以后，整个消息就变成下面这样（十六进制格式）。

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000018

如果原始的消息长度超过了 512，需要将它补成 512 的倍数。然后把整个消息分成一个一个 512 位的数据块，分别处理每一个数据块，从而得到消息摘要。

（3）使用的常量

一系列的常量字 K(0)、K(1)、…、K(79)，如果以十六进制给出。它们如下所示。

Kt = 0x5A827999 (0 <= t <= 19)

Kt = 0x6ED9EBA1 (20 <= t <= 39)

Kt = 0x8F1BBCDC (40 <= t <= 59)

Kt = 0xCA62C1D6 (60 <= t <= 79).

（4）需要使用的函数

在 SHA1 中需要一系列的函数。每个函数 ft (0 <= t <= 79)都操作 32 位字 B、C、D 并且产生 32 位字作为输出。ft(B,C,D)可以如下定义。

ft(B,C,D) = (B AND C) or ((NOT B) AND D) (0 <= t <= 19)

ft(B,C,D) = B XOR C XOR D (20 <= t <= 39)

ft(B,C,D) = (B AND C) or (B AND D) or (C AND D) (40 <= t <= 59)

ft(B,C,D) = B XOR C XOR D (60 <= t <= 79)

（5）计算消息摘要

必须使用进行了补位和补长度后的消息来计算消息摘要。计算需要两个缓冲区，每个都由 5 个 32 位的字组成，还需要一个 80 个 32 位字的缓冲区。第一个 5 个字的缓冲区被标识为 A、B、C、D、E。第一个 5 个字的缓冲区被标识为 H0、H1、H2、H3、H4。80 个字的缓冲区被标识为 W0、W1、…、W79。

另外，还需要一个 1 个字的 TEMP 缓冲区。

为了产生消息摘要，在第 4 部分中定义的 16 个字的数据块 M1、M2、…、M*n* 会依次进行处理，处理每个数据块 M*i* 包含 80 个步骤。

在处理每个数据块之前，缓冲区{Hi}被初始化为下面的值（十六进制）。

H0 = 0x67452301

H1 = 0xEFCDAB89

H2 = 0x98BADCFE

H3 = 0x10325476

H4 = 0xC3D2E1F0

现在开始处理 M1、M2、…、M*n*。为了处理 M*i*，需要进行下面的步骤。

① 将 M*i* 分成 16 个字 W0、W1、…、W15、W0 是最左边的字。

② 对于 *t*=16 到 79，令 Wt = S1(Wt-3 XOR Wt-8 XOR Wt- 14 XOR Wt-16)。

③令 A = H0、B = H1、C = H2、D = H3、E = H4。

④ 对于 *t* = 0 到 79，执行下面的循环。

TEMP = S5(A) + ft(B,C,D) + E + Wt + Kt;

E = D; D = C; C = S30(B); B = A; A = TEMP;

⑤ 令 H0 = H0 + A、H1 = H1 + B、H2 = H2 + C、H3 = H3 + D、H4 = H4 + E。

在处理完所有的 M*n* 后，消息摘要是一个 160 位的字符串，以下面的顺序标识。

H0 H1 H2 H3 H4。

实现过程

例 15.22　SHA 加密（光盘位置：光盘\MR\Instance\15\22）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、stdlib.h、string.h。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

（3）初始化消息摘要缓冲区。形式参数 hd 是 SHA1_CONTEXT 结构变量的地址。代码如下。

```
void sha1_init(SHA1_CONTEXT *hd)
{
    hd->h0 = 0x67452301;
    hd->h1 = 0xefcdab89;
    hd->h2 = 0x98badcfe;
    hd->h3 = 0x10325476;
    hd->h4 = 0xc3d2e1f0;
    hd->nblocks = 0;
    hd->count = 0;
}
```

（4）转换消息摘要。形式参数 hd 是 SHA1_CONTEXT 结构变量的地址；参数 data 为缓冲区地址。代码如下。

```
static void transform(SHA1_CONTEXT *hd, unsigned char *data)
{
    u32 a, b, c, d, e, tm;
    u32 x[16];                                          //512 位的块
    a = hd->h0;                                         //记录初始缓冲区内容
    b = hd->h1;
```

```
    c = hd->h2;
    d = hd->h3;
    e = hd->h4;

    #ifdef BIG_ENDIAN_HOST
        memcpy(x, data, 64);                                    //复制块
    #else
        {
            int i;                                              //声明整型变量
            unsigned char *p2;                                  //声明字符指针变量
            for (i = 0, p2 = (unsigned char*)x; i < 16; i++, p2 += 4)
            {
                p2[3] =  *data++;
                p2[2] =  *data++;
                p2[1] =  *data++;
                p2[0] =  *data++;
            }
        }
    #endif


    #define K1  0x5A827999L                                     //定义宏
    #define K2  0x6ED9EBA1L
    #define K3  0x8F1BBCDCL
    #define K4  0xCA62C1D6L
    #define F1(x,y,z)   ( z ^ ( x & ( y ^ z ) ) )               //等价于(xy)+(/x*z)
    #define F2(x,y,z)   ( x ^ y ^ z )
    #define F3(x,y,z)   ( ( x & y ) | ( z & ( x | y ) ) )
    #define F4(x,y,z)   ( x ^ y ^ z )


    #define M(i) ( tm =   x[i&0x0f] ^ x[(i-14)&0x0f]    \
    ^ x[(i-8)&0x0f] ^ x[(i-3)&0x0f]      \
    , (x[i&0x0f] = rol(tm,1)) )                             //计算从 16～79 的 Wt

    #define R(a,b,c,d,e,f,k,m)  do { e += rol( a, 5 )   \
    + f( b, c, d )                              \
    + k                                         \
    + m;                                        \

    } while(0)
    R(a, b, c, d, e, F1, K1, x[0]);
    R(e, a, b, c, d, F1, K1, x[1]);
     \
        *Â Ô * \
        R(c, d, e, a, b, F4, K4, M(78));
    R(b, c, d, e, a, F4, K4, M(79));

    /* Update chaining vars */
    hd->h0 += a;
    hd->h1 += b;
    hd->h2 += c;
    hd->h3 += d;
    hd->h4 += e;
}
```

（5）更新消息摘要。形式参数 hd 是 SHA1_CONTEXT 结构变量的地址；参数 data 为缓冲区地址；参数 inlen 用于指定更新长度。代码如下。

```
static void sha1_write(SHA1_CONTEXT *hd, unsigned char *inbuf, size_t inlen)
{
```

```
    if (hd->count == 64)
    {
         /* flush the buffer */
        transform(hd, hd->buf);
        hd->count = 0;
        hd->nblocks++;
    }
    if (!inbuf)
        return ;
    if (hd->count)
    {
        for (; inlen && hd->count < 64; inlen--)
            hd->buf[hd->count++] =  *inbuf++;
        sha1_write(hd, NULL, 0);
        if (!inlen)
            return ;
    }

    while (inlen >= 64)
    {
        transform(hd, inbuf);
        hd->count = 0;
        hd->nblocks++;
        inlen -= 64;
        inbuf += 64;
    }
    for (; inlen && hd->count < 64; inlen--)
        hd->buf[hd->count++] =  *inbuf++;
}
```

（6）完成计算，返回摘要。代码如下。

```
static void sha1_final(SHA1_CONTEXT *hd) //sha-1 算法实现
{
    u32 t, msb, lsb;
    unsigned char *p;

    sha1_write(hd, NULL, 0); /* flush */;

    t = hd->nblocks;
    /* multiply by 64 to make a byte count */
    lsb = t << 6;
    msb = t >> 26;
    /* add the count */
    t = lsb;
    if ((lsb += hd->count) < t)
        msb++;
    /* multiply by 8 to make a bit count */
    t = lsb;
    lsb <<= 3;
    msb <<= 3;
    msb |= t >> 29;

    if (hd->count < 56)
    {
         /* enough room */
        hd->buf[hd->count++] = 0x80; /* pad */
        while (hd->count < 56)
            hd->buf[hd->count++] = 0;
         /* pad */
    }
```

```
    else
    {
         /* need one extra block */
        hd->buf[hd->count++] = 0x80; /* pad character */
        while (hd->count < 64)
            hd->buf[hd->count++] = 0;
        sha1_write(hd, NULL, 0); /* flush */;
        memset(hd->buf, 0, 56); /* fill next block with zeroes */
    }
    /* append the 64 bit count */
    hd->buf[56] = msb >> 24;
    hd->buf[57] = msb >> 16;
    hd->buf[58] = msb >> 8;
    hd->buf[59] = msb;
    hd->buf[60] = lsb >> 24;
    hd->buf[61] = lsb >> 16;
    hd->buf[62] = lsb >> 8;
    hd->buf[63] = lsb;
    transform(hd, hd->buf);

    p = hd->buf;
    #ifdef BIG_ENDIAN_HOST
        #define X(a) do { *(u32*)p = hd->h##a ; p += 4; } while(0)
    #else /* little endian */
        #define X(a) do { *p++ = hd->h##a >> 24; *p++ = hd->h##a >> 16; \
        *p++ = hd->h##a >> 8; *p++ = hd->h##a; } while(0)
    #endif
    X(0);
    X(1);
    X(2);
    X(3);
    X(4);
    #undef X
}
```

（7）在主函数 main()中，打开一个文本文件，并调用函数 sha1_init()、sha1_write()、sha1_final()对文本文件内容进行加密，并输出到屏幕上。代码如下。

```
int main(int argc, char **argv)
{
    if (argc < 2)
    {
        fprintf(stderr, "usage: shalsum filenames\n");
        exit(1);
    }
    for (argc--, argv++; argc; argv++, argc--)
    {
        FILE *fp;
        char buffer[4096];
        size_t n;
        SHA1_CONTEXT ctx;
        int i;

        fp = fopen(*argv, "rb");
        if (!fp)
        {
            fprintf(stderr, "can't open `%s': %s\n",  *argv, strerror(errno));
            exit(1);
        }
        sha1_init(&ctx);
        while ((n = fread(buffer, 1, sizeof buffer, fp)))
```

```
            sha1_write(&ctx, buffer, n);
        if (ferror(fp))
        {
            fprintf(stderr, "error reading `%s': %s\n",  *argv, strerror(errno))
                ;
            exit(1);
        }
        sha1_final(&ctx);
        fclose(fp);

        for (i = 0; i < 20; i++)
            printf("%02x", ctx.buf[i]);
        printf("  %s\n",  *argv);
    }
    return 0;
}
```

举一反三

根据本实例，读者可以：

- 采用 SHA1 算法制作软件授权文件；
- 采用 SHA1 算法加密用户登录密码。

算法 23　MD5 加密

实例说明

MD5（Message-Digest Algorithm 5）是一种被广泛使用的“消息-摘要算法”。“消息-摘要算法”实际上就是一个单项散列函数。数据块经过单向散列函数得到一个固定长度的散列值，数据块的签名就是计算数据块的散列值，MD5 算法的散列值为 128 位。本实例用于介绍对字符串进行 MD5 加密的方法，运行本实例编译后的可执行文件，运行参数为 swordfish，运行结果如图 15.29 所示。

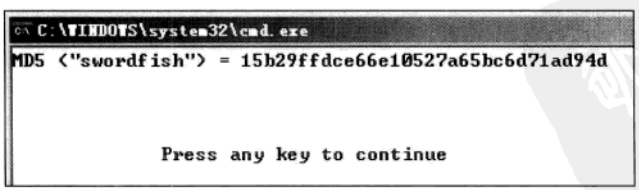


图 15.29　MD5 加密效果

技术要点

在 MD5 算法中，首先需要对信息进行填充，使其字节长度对 512 求余的结果等于 448。因此，

信息的字节长度（Bits Length）将被扩展至 *N**512+448，即 *N**64+56 个字节（Bytes），*N* 为一个正整数。填充的方法如下：在信息的后面填充一个 1 和若干个 0，直到满足上面的条件时才停止用 0 对信息的填充；然后，在这个结果后面附加一个以 64 位二进制表示的填充前信息长度。经过这两步的处理，现在的信息字节长度=*N**512+448+64=(*N*+1)*512，即长度恰好是 512 的整数倍。这样做的原因是为满足后面处理中对信息长度的要求。

MD5 中有 4 个 32 位被称作链接变量（Chaining Variable）的整数参数，它们分别为 A=0x01234567、B=0x89abcdef、C=0xfedcba98、D=0x76543210。

当设置好这 4 个链接变量后，就开始进入算法的 4 轮循环运算。循环的次数是信息中 512 位信息分组的数目。

将上面 4 个链接变量复制到另外 4 个变量中：A 到 a、B 到 b、C 到 c、D 到 d。

主循环有 4 轮，每轮循环都很相似。第一轮进行 16 次操作。每次操作对 a、b、c 和 d 中的其中 3 个作一次非线性函数运算，然后将所得结果加上第 4 个变量，文本的一个子分组和一个常数。再将所得结果向右环移一个不定的数，并加上 a、b、c 或 d 中之一。最后用该结果取代 a、b、c 或 d 中之一。

以一下是每次操作中用到的 4 个非线性函数（每轮一个）。

F(X,Y,Z) =(X&Y)|((～X)&Z)

G(X,Y,Z) =(X&Z)|(Y&(～Z))

H(X,Y,Z) =X^Y^Z

I(X,Y,Z)=Y^(X|(～Z))

（&是与、|是或、～是非、^是异或）

这 4 个函数的说明：如果 X、Y 和 Z 的对应位是独立和均匀的，那么结果的每一位也应是独立和均匀的。F 是一个逐位运算的函数。即，如果 X，那么 Y，否则 Z。函数 H 是逐位奇偶操作符。

假设 M*j* 表示消息的第 *j* 个子分组（从 0～15）

FF(a,b,c,d,Mj,s,ti)表示 a=b+((a+(F(b,c,d)+Mj+ti)，

GG(a,b,c,d,Mj,s,ti)表示 a=b+((a+(G(b,c,d)+Mj+ti)，

HH(a,b,c,d,Mj,s,ti)表示 a=b+((a+(H(b,c,d)+Mj+ti)，

II(a,b,c,d,Mj,s,ti)表示 a=b+((a+(I(b,c,d)+Mj+ti)，

这 4 轮（64 步）如下所示

（1）第一轮

FF(a,b,c,d,M0,7,0xd76aa478)

FF(d,a,b,c,M1,12,0xe8c7b756)

FF(c,d,a,b,M2,17,0x242070db)

FF(b,c,d,a,M3,22,0xc1bdceee)

FF(a,b,c,d,M4,7,0xf57c0faf)

FF(d,a,b,c,M5,12,0x4787c62a)

FF(c,d,a,b,M6,17,0xa8304613)

FF(b,c,d,a,M7,22,0xfd469501)

FF(a,b,c,d,M8,7,0x698098d8)

FF(d,a,b,c,M9,12,0x8b44f7af)

FF(c,d,a,b,M10,17,0xffff5bb1)

FF(b,c,d,a,M11,22,0x895cd7be)

FF(a,b,c,d,M12,7,0x6b901122)

FF(d,a,b,c,M13,12,0xfd987193)

FF(c,d,a,b,M14,17,0xa679438e)

FF(b,c,d,a,M15,22,0x49b40821)

（2）第二轮

GG(a,b,c,d,M1,5,0xf61e2562)

GG(d,a,b,c,M6,9,0xc040b340)

GG(c,d,a,b,M11,14,0x265e5a51)

GG(b,c,d,a,M0,20,0xe9b6c7aa)

GG(a,b,c,d,M5,5,0xd62f105d)

GG(d,a,b,c,M10,9,0x02441453)

GG(c,d,a,b,M15,14,0xd8a1e681)

GG(b,c,d,a,M4,20,0xe7d3fbc8)

GG(a,b,c,d,M9,5,0x21e1cde6)

GG(d,a,b,c,M14,9,0xc33707d6)

GG(c,d,a,b,M3,14,0xf4d50d87)

GG(b,c,d,a,M8,20,0x455a14ed)

GG(a,b,c,d,M13,5,0xa9e3e905)

GG(d,a,b,c,M2,9,0xfcefa3f8)

GG(c,d,a,b,M7,14,0x676f02d9)

GG(b,c,d,a,M12,20,0x8d2a4c8a)

（3）第三轮

HH(a,b,c,d,M5,4,0xfffa3942)

HH(d,a,b,c,M8,11,0x8771f681)

HH(c,d,a,b,M11,16,0x6d9d6122)

HH(b,c,d,a,M14,23,0xfde5380c)

HH(a,b,c,d,M1,4,0xa4beea44)

HH(d,a,b,c,M4,11,0x4bdecfa9)

HH(c,d,a,b,M7,16,0xf6bb4b60)

HH(b,c,d,a,M10,23,0xbebfbc70)

HH(a,b,c,d,M13,4,0x289b7ec6)

HH(d,a,b,c,M0,11,0xeaa127fa)

HH(c,d,a,b,M3,16,0xd4ef3085)

HH(b,c,d,a,M6,23,0x04881d05)

HH(a,b,c,d,M9,4,0xd9d4d039)

HH(d,a,b,c,M12,11,0xe6db99e5)

HH(c,d,a,b,M15,16,0x1fa27cf8)

HH(b,c,d,a,M2,23,0xc4ac5665)

（4）第四轮

II(a,b,c,d,M0,6,0xf4292244)

II(d,a,b,c,M7,10,0x432aff97)

II(c,d,a,b,M14,15,0xab9423a7)

II(b,c,d,a,M5,21,0xfc93a039)

II(a,b,c,d,M12,6,0x655b59c3)

II(d,a,b,c,M3,10,0x8f0ccc92)

II(c,d,a,b,M10,15,0xffeff47d)

II(b,c,d,a,M1,21,0x85845dd1)

II(a,b,c,d,M8,6,0x6fa87e4f)

II(d,a,b,c,M15,10,0xfe2ce6e0)

II(c,d,a,b,M6,15,0xa3014314)

II(b,c,d,a,M13,21,0x4e0811a1)

II(a,b,c,d,M4,6,0xf7537e82)

II(d,a,b,c,M11,10,0xbd3af235)

II(c,d,a,b,M2,15,0x2ad7d2bb)

II(b,c,d,a,M9,21,0xeb86d391)

常数 t*i* 可以如下选择。

在第 *i* 步中，t*i* 是 4294967296*abs(sin(*i*))的整数部分，*i* 的单位是弧度（4294967296 等于 2 的 32 次方）。

所有这些完成之后，将 A、B、C、D 分别加上 a、b、c、d。然后用下一分组数据继续运行算法，最后的输出是 A、B、C 和 D 的级联。

实现过程

例 15.23　MD5 加密（光盘位置：光盘\MR\Instance\15\23）。

（1）在 TC 中创建一个 C 文件。

（2）创建头文件。

① 创建头文件，命名为 global.h，用于定义宏和类型别名。代码如下。

```
#ifndef _GLOBAL_H_
#define _GLOBAL_H_                                    //定义宏
#ifndef PROTOTYPES
#define PROTOTYPES 0                                  //定义宏
#endif
typedef unsigned char *POINTER;                       //无符号字符指针类型别名
typedef unsigned short int UINT2;                     //无符号短整型变量别名
typedef unsigned long int UINT4;                      //无符号长整型变量别名
#if PROTOTYPES
    #define PROTO_LIST(list) list                     //定义宏
#else
    #define PROTO_LIST(list) ()                       //定义宏
#endif
#endif
```

② 创建头文件，命名为 md5.h，用于定义结构和函数声明。代码如下。

```
#ifndef _MD5_H_
#define _MD5_H_                                                    //定义宏
typedef struct
{
    UINT4 state[4];
    UINT4 count[2];
    unsigned char buffer[64];
} MD5_CTX;                                                         //定义结构 MD5_CTX
void MD5Init PROTO_LIST ((MD5_CTX *));                             //声明函数
void MD5Update PROTO_LIST((MD5_CTX *, unsigned char *, unsigned int));//声明函数
void MD5Final PROTO_LIST ((unsigned char [16], MD5_CTX *));        //声明函数
#endif
```

（3）引用头文件 stdio.h、string.h、global.h、md5.h。

```
#include <stdio.h>
#include <string.h>
#include "global.h"
#include "md5.h"
```

（4）MD5 主要加密过程。

① 定义 4 个非线性函数，代码如下。

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))
```

② FF(a,b,c,d,x,s,ac)表示 a=b+((a+(F(b,c,d)+Mj+ti)<<<s)；GG(a,b,c,d,x,s,ac)表示 a=b+((a+(G(b,c,d)+Mj+ti)<<<s)；HH(a,b,c,d,x,s,ac)表示 a=b+((a+(H(b,c,d)+Mj+ti)<<<s)；II(a,b,c,d,x,s,ac)表示 a=b+((a+(I(b,c,d)+Mj+ti)<<<s)，代码如下。

```
#define FF(a, b, c, d, x, s, ac) { (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); (a) = ROTATE_LEFT ((a),
(s)); (a) += (b);}

#define GG(a, b, c, d, x, s, ac) { \
(a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}

#define HH(a, b, c, d, x, s, ac) { \
(a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}

#define II(a, b, c, d, x, s, ac) { \
(a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}
```

③ MD5 初始化，为 A、B、C、D 这 4 个 32 位变量赋值，代码如下。

```
void MD5Init(context)MD5_CTX *context;
{
    context->count[0] = context->count[1] = 0;
    context->state[0] = 0x67452301;
    context->state[1] = 0xefcdab89;
    context->state[2] = 0x98badcfe;
    context->state[3] = 0x10325476;
}
```

④ MD5 块更新。继续 MD5 消息摘要操作，进行算法的 4 轮循环运算。代码如下。

```
void MD5Update(context, input, inputLen)MD5_CTX *context;
unsigned char *input;
unsigned int inputLen;
{
    unsigned int i, index, partLen;
    index = (unsigned int)((context->count[0] >> 3) &0x3F);
    if ((context->count[0] += ((UINT4)inputLen << 3)) < ((UINT4)inputLen << 3))
        context->count[1]++;
    context->count[1] += ((UINT4)inputLen >> 29);
    partLen = 64-index;
    if (inputLen >= partLen)
    {
        MD5_memcpy((POINTER) &context->buffer[index], (POINTER)input, partLen);
        MD5Transform(context->state, context->buffer);
        for (i = partLen; i + 63 < inputLen; i += 64)
            MD5Transform(context->state, &input[i]);
        index = 0;
    }
    else
        i = 0;
    MD5_memcpy((POINTER) &context->buffer[index], (POINTER) &input[i], inputLen
        - i);
}
```

⑤ MD5 完成。结束 MD5 消息摘要操作，写消息摘要，进行 MD5_CTX 结构变量的清零操作。代码如下。

```
void MD5Final(digest, context)unsigned char digest[16];
MD5_CTX *context;
{
    unsigned char bits[8];
    unsigned int index, padLen;
    Encode(bits, context->count, 8);
    index = (unsigned int)((context->count[0] >> 3) &0x3f);
    padLen = (index < 56) ? (56-index): (120-index);
    MD5Update(context, PADDING, padLen);
    MD5Update(context, bits, 8);
    Encode(digest, context->state, 16);
    MD5_memset((POINTER)context, 0, sizeof(*context));
}
```

⑥ MD5 基本转换过程。参数 state 用于传递 A、B、C、D 值，参数 block 用于传递字符类型的数组，代码如下。

```
static void MD5Transform(UINT4 state[4], unsigned char block[64])
{
    UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];
    Decode(x, block, 64);
    FF(a, b, c, d, x[0], S11, 0xd76aa478); /* 1 */
    FF(d, a, b, c, x[1], S12, 0xe8c7b756); /* 2 */
    FF(c, d, a, b, x[2], S13, 0x242070db); /* 3 */
    FF(b, c, d, a, x[3], S14, 0xc1bdceee); /* 4 */
    FF(a, b, c, d, x[4], S11, 0xf57c0faf); /* 5 */
    FF(d, a, b, c, x[5], S12, 0x4787c62a); /* 6 */
    FF(c, d, a, b, x[6], S13, 0xa8304613); /* 7 */
    FF(b, c, d, a, x[7], S14, 0xfd469501); /* 8 */
    FF(a, b, c, d, x[8], S11, 0x698098d8); /* 9 */
    FF(d, a, b, c, x[9], S12, 0x8b44f7af); /* 10 */
    FF(c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
    FF(b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
    FF(a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
    FF(d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
    FF(c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
    FF(b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

    /* Round 2 */
    GG(a, b, c, d, x[1], S21, 0xf61e2562); /* 17 */
    GG(d, a, b, c, x[6], S22, 0xc040b340); /* 18 */
    GG(c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
    GG(b, c, d, a, x[0], S24, 0xe9b6c7aa); /* 20 */
    GG(a, b, c, d, x[5], S21, 0xd62f105d); /* 21 */
    GG(d, a, b, c, x[10], S22, 0x2441453); /* 22 */
    GG(c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
    GG(b, c, d, a, x[4], S24, 0xe7d3fbc8); /* 24 */
    GG(a, b, c, d, x[9], S21, 0x21e1cde6); /* 25 */
    GG(d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
    GG(c, d, a, b, x[3], S23, 0xf4d50d87); /* 27 */

    GG(b, c, d, a, x[8], S24, 0x455a14ed); /* 28 */
    GG(a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
    GG(d, a, b, c, x[2], S22, 0xfcefa3f8); /* 30 */
    GG(c, d, a, b, x[7], S23, 0x676f02d9); /* 31 */
    GG(b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

    /* Round 3 */
    HH(a, b, c, d, x[5], S31, 0xfffa3942); /* 33 */
```

```
    HH(d, a, b, c, x[8], S32, 0x8771f681); /* 34 */
    HH(c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
    HH(b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
    HH(a, b, c, d, x[1], S31, 0xa4beea44); /* 37 */
    HH(d, a, b, c, x[4], S32, 0x4bdecfa9); /* 38 */
    HH(c, d, a, b, x[7], S33, 0xf6bb4b60); /* 39 */
    HH(b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
    HH(a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
    HH(d, a, b, c, x[0], S32, 0xeaa127fa); /* 42 */
    HH(c, d, a, b, x[3], S33, 0xd4ef3085); /* 43 */
    HH(b, c, d, a, x[6], S34, 0x4881d05); /* 44 */
    HH(a, b, c, d, x[9], S31, 0xd9d4d039); /* 45 */
    HH(d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
    HH(c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
    HH(b, c, d, a, x[2], S34, 0xc4ac5665); /* 48 */

    /* Round 4 */
    II(a, b, c, d, x[0], S41, 0xf4292244); /* 49 */
    II(d, a, b, c, x[7], S42, 0x432aff97); /* 50 */
    II(c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
    II(b, c, d, a, x[5], S44, 0xfc93a039); /* 52 */
    II(a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
    II(d, a, b, c, x[3], S42, 0x8f0ccc92); /* 54 */
    II(c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
    II(b, c, d, a, x[1], S44, 0x85845dd1); /* 56 */
    II(a, b, c, d, x[8], S41, 0x6fa87e4f); /* 57 */
    II(d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
    II(c, d, a, b, x[6], S43, 0xa3014314); /* 59 */
    II(b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
    II(a, b, c, d, x[4], S41, 0xf7537e82); /* 61 */
    II(d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
    II(c, d, a, b, x[2], S43, 0x2ad7d2bb); /* 63 */
    II(b, c, d, a, x[9], S44, 0xeb86d391); /* 64 */

    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
    MD5_memset((POINTER)x, 0, sizeof(x));
}
```

⑦ 输出编码。代码如下。

```
static void Encode(output, input, len)unsigned char *output;
UINT4 *input;
unsigned int len;
{
    unsigned int i, j;
    for (i = 0, j = 0; j < len; i++, j += 4)
    {
        output[j] = (unsigned char)(input[i] &0xff);
        output[j + 1] = (unsigned char)((input[i] >> 8) &0xff);
        output[j + 2] = (unsigned char)((input[i] >> 16) &0xff);
        output[j + 3] = (unsigned char)((input[i] >> 24) &0xff);
    }
}
```

⑧ 输出译码。代码如下。

```
static void Decode(output, input, len)UINT4 *output;
unsigned char *input;
```

```
unsigned int len;
{
    unsigned int i, j;
    for (i = 0, j = 0; j < len; i++, j += 4)
        output[i] = ((UINT4)input[j]) | (((UINT4)input[j + 1]) << 8) | (((UINT4)
            input[j + 2]) << 16) | (((UINT4)input[j + 3]) << 24);
}
```

⑨ 获取 MD5 消息摘要并进行输出，代码如下。

```
static void MDString(char *string)
{
    MD_CTX context;
    unsigned char digest[16];
    unsigned int len = strlen(string);

    MDInit(&context);
    MDUpdate(&context, string, len);
    MDFinal(digest, &context);

    printf("MD%d (\"%s\") = ", MD, string);
    MDPrint(digest);
    printf("\n");
}

static void MDPrint(unsigned char digest[16])
{
    unsigned int i;
    for (i = 0; i < 16; i++)
        printf("%02x", digest[i]);

}
```

⑩ 在主函数 main()中调用函数 MDString()，对传入程序的字符串参数进行加密，并将加密后的结果输出到计算机屏幕。代码如下。

```
int main(int argc, char *argv[])
{
    int i;
    if (argc > 1)
    {
        MDString(argv[1]);
        return (0);
    }
}
```

举一反三

根据本实例，读者可以：

- 使用 MD5 算法的文本加密程序；
- 使用 MD5 算法获取文件信息摘要。

算法 24　RSA 加密

实例说明

RAS 算法是非对称加密的代表。RSA 算法是第一个同时用于加密和数字签名的算法，易于理解和操作。本实例用于介绍 RSA 加密的方法，运行本实例编译后的可执行文件，指定 p 的值为 5，q 的值为 11，e 的值为 3，选择 1，指定 m 的值 14，程序输出结果如图 15.30 所示。

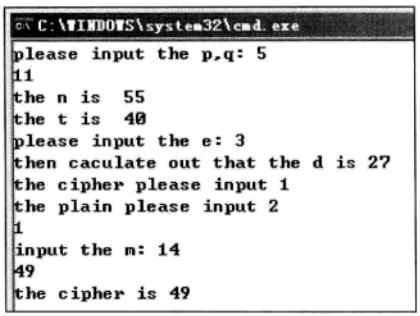


图 15.30　RSA 算法输出结果

技术要点

RSA 算法是一种非对称密码算法，所谓非对称，就是指该算法需要一组密钥，其中包括公钥和私钥。RSA 加密或解密步骤如下。

（1）获取公钥与私钥

① 随意选择两个大的质数 p 和 q，p 不等于 q，计算 $n=pq$。

② 计算不大于 n 且与 n 互质的整数个数 f，公式为 $f=(p-1)(q-1)$。

③ 选择一个整数 e 与 $(p-1)(q-1)$ 互质，并且 e 小于 $(p-1)(q-1)$。

④ 计算 d，公式：$de \bmod (p-1)(q-1)=1$。

⑤ d 和 n 是私钥，p、q、e 是公钥。

（2）加密过程

设 e 为加密密钥，明文为 m，密文为 c，则加密公式如下。

$$c = (m \wedge e) \bmod n$$

（3）解密过程

设 e 为加密密钥，明文为 m，密文为 c，d 为解密密钥，则解密公式如下。

$$m = (c \wedge d) \bmod n$$

实现过程

例 15.24　RSA 加密（光盘位置：光盘\MR\Instance\15\24）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h。

```
#include <stdio.h>
```

（3）创建自定义函数 candp()，用于进行加密，以获取密文，或进行解密，以获取明文，代码如下。

```
int candp(int a, int b, int c)
{
    int r = 1;                                  //声明整型变量并赋值 1
    b = b + 1;                                  //b 自加 1
    while (b != 1)                              //当 b 不等于 1 保持循环
    {
        r = r * a;                              //r 自乘 a
        r = r % c;                              //r 自模 c
        b--;                                    //b 自减
    }
    printf("%d\n", r);                          //输出 r 值
    return r;
}
```

（4）输入 p、q、e 的数值。根据输入的数值计算 p 与 q 的乘积 n，并调用函数 candp()进行加密或解密操作。代码如下。

```
void main()
{
    int p, q, e, d, m, n;                       //声明整型变量
    char s;                                     //声明字符变量
    printf("please input the p,q: ");           //提示请输入 p，q
    scanf("%d%d", &p, &q);                      //带格式输入 p、q 值
    n = p * q;                                  //计算 p 与 q 的积
    printf("the n is %3d\n", n);                //输出字符串
    t = (p - 1)*(q - 1);                        //计算(p-1)*(q-1)的乘积
    printf("the t is %3d\n", t);                //输出字符串
    printf("please input the e: ");             //提示输入 e
    scanf("%d", &e);                            //带格式输入 e
    if (e < 1 || e > t)                         //当 e 小于 1 或 e 大于 t
    {
        printf("e is error,please input again: ");  //提示 e 值错误，请重新输入
        scanf("%d", &e);                        //带格式输入 e
    }
    d = 1;                                      //为 d 赋值 1
    while (((e *d) % t) != 1)                   //表达式结果不等于 1 时
        d++;                                    //d 自加
    printf("then caculate out that the d is %d\n", d);  //输出字符串
    printf("the cipher please input 1\n");      //输出字符串
    printf("the plain please input 2\n");       //输出字符串
    scanf("%d", &r);                            //带格式输入 r
    switch (r)                                  //选择
    {
        case 1:                                 //r 值为 1 时
            printf("input the m: ");  /*输入要加密的密文数字*/  //输出字符串
            scanf("%d", &m);                    //带格式输入 m
            c = candp(m, e, n);                 //进行加密获取密文
            printf("the cipher is %d\n", c);    //输出 c 值
            break;
        case 2:
```

```
            printf("input the c: "); /*输入要解密的密文数字*/
            scanf("%d", &c);                                    //带格式输入 c
            m = candp(c, d, n);                                 //进行解密获取明文
            printf("the cipher is %d\n", m);                    //输出 m 值
            break;
        }
        getch();
    }
```

举一反三

根据本实例，读者可以：

- 采用 RSA 算法对字符串进行加密；
- 采用 RSA 算法制作注册机程序。

算法 25　DES 加密

实例说明

美国数据加密标准 DES 是一种对称加密算法。对称是指采用保密密钥既可用于加密也可用于解密。DES 的算法是公开的，密码由用户自己保护。密钥长度为 64 位，其中有 8 位奇偶校验，有效长度为 56 位，即采用一个 56 位的有效密钥对 64 位的数据块进行加密。本实例用于介绍 DES 加密的方法，运行本实例编译后的可执行文件，输入字符串 swordfish，输入密码 aj，运行效果如图 15.31 所示。

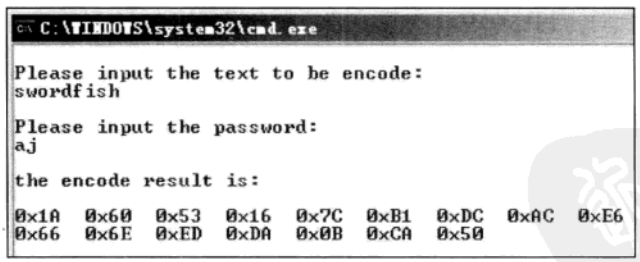


图 15.31　DES 加密效果

技术要点

1. 取得密钥

从用户处取得一个 64 位（本文如未特指，均指二进制位）长的密码 key，如下所示。

$$key=k_1k_2k_3\ldots k_{65}k_{64}$$

去除 64 位密码中作为奇偶校验位的第 8、16、24、32、40、48、56、64 位，剩下的 56 位作为有效输入密钥。

2. 等分密钥

将在上步中生成的 56 位输入密钥分成均等的 A、B 两部分，每部分为 28 位，参照图 15.32 和图 15.33 把输入密钥的位值填入相应的位置.。按照图 10.8 所示，A 的第一位为输入的 64 位密钥的第 57 位，A 的第 2 位为 64 位密钥的第 49 位，依此类推，A 的最后一位最后一位是 64 位密钥的第 36 位。

$$k=k_1k_2k_3\ldots k_{55}k_{56}$$

$$A=k_{57}k_{49}k_{41}\ldots k_{44}k_{36}$$

$$B=k_{65}k_{55}k_{47}\ldots k_{12}k_4$$

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	50	44	36

图 15.32　密钥 A 部分

65	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

图 15.33　密钥 B 部分

3. 密钥移位

DES 算法的密钥是经过 16 次迭代得到一组密钥的，把在上步中生成的 A、B 视为迭代的起始密钥，如图 15.34 所示，在第 *i* 次迭代时密钥循环左移的位数。比如在第 1 次迭代时密钥循环左移 1 位，第 3 次迭代时密钥循环左移 2 位。

i	1	2	3	4	5	6	7	8
ǿ	1	1	2	2	2	2	2	2
i	9	10	11	12	13	14	15	16
ǿ	1	2	2	2	2	2	2	1

图 15.34　第 *i* 次迭代时密钥循环左移的位数

第 9 次迭代时密钥循环左移 1 位，第 14 次迭代时密钥循环左移 2 位。

第一次迭代：

$$A(1) = ǿ(1)\,A$$

$$B(1) = ǿ(1)\,B$$

第 i 次迭代：

$$A(i) = ǿ(i)\,A(i\text{-}1)$$

$$B(i) = ǿ(i)\,B(i\text{-}1)$$

4. 选取密钥

在上步中第 i 次迭代生成的两个 28 位长的密钥为 $A^{(i)},B^{(i)}$。

$$A^{(i)}=A^{(i)}_1A^{(i)}_2A^{(i)}_3\ldots A^{(i)}_{27}A^{(i)}_{28}$$

$$B^{(i)}=B^{(i)}_1B^{(i)}_2B^{(i)}_3\ldots B^{(i)}_{27}B^{(i)}_{28}$$

把 $A^{(i)},B^{(i)}$合并

$$C^{(i)}=A^{(i)}B^{(i)}=A^{(i)}_1A^{(i)}_2A^{(i)}_3\ldots A^{(i)}_{27}A^{(i)}_{28}B^{(i)}_1B^{(i)}_2B^{(i)}_3\ldots B^{(i)}_{27}B^{(i)}_{28}$$

$$=C^{(i)}_1C^{(i)}_2C^{(i)}_3\ldots C^{(i)}_{47}C^{(i)}_{48}$$

按照图 15.35 所示，k 的第一位为 56 位密钥的第 14 位，k 的第 2 位为 56 位密钥的第 17 位，……依此类推，k 的最后一位是 56 位密钥的第 32 位。生成与进行第 i 次迭代加密的数据进行按位异或的 48 位使用密钥。

14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

图 15.35　k的每一位与密钥位对应图

$$k^{(i)}=C^{(i)}_{14}C^{(i)17}C^{(i)}_{11}\ldots C^{(i)}_{29}C^{(i)}_{32}$$

$$=k^{(i)}_1k^{(i)}_2k^{(i)}_3\ldots k^{(i)}_{47}k^{(i)}_{48}$$

5. 迭代

DES 算法密钥生成需要进行 16 次迭代，在完成 16 次迭代前，循环执行 3、4 步，最终形成 16 套加密密钥：key[0]、key[1]、key[2]、……、key[14]、key[15]。

实现过程

例 15.25　DES 加密（光盘位置：光盘\MR\Instance\15\25）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、string.h。

```
#include "stdio.h"
#include "string.h"
```

（3）创建函数 des()用于生成加密密钥，把待加密的明文数据分割成 64 位的块，逐块完成 16 次迭代加密，密文存放在 data 所指向的内存中。第一个形式参数用于存放待加密明文的内存指针；第二个形式参数用于存放用户输入的密钥内存的指针；第三个形式参数为待加密明文的长度。函数代码如下。

```
INT32 des(ULONG8 *data, ULONG8 *key, INT32 readlen)
{
    INT32 i = 0;                                    //声明整型变量
    makefirstkey((ULONG32*)key);                    /*产生密钥*/
```

```
    for (i = 0; i < readlen; i += 8)                    //循环
    {
        handle_data((ULONG32*) &data[i], DESENCRY);     //调用 handle_data 函数
    }
    return SUCCESS;                                     //返回 SUCCESS
}
```

（4）创建函数 Ddes()用于生成解密密钥，把待解密文分割成 64 位的块，逐块完成 16 次迭代解密，解密后的明文存放在 data 所指向的内存中。第一个形式参数用于存放待解密文的内存指针；第二个形式参数用于存放用户输入的密钥内存的指针；第三个形式参数为待解密文的长度。函数代码如下。

```
INT32 Ddes(ULONG8 *data, ULONG8 *key, INT32 readlen)
{
    INT32 i = 0;                                        //声明整型变量
    makefirstkey((ULONG32*)key);                        //调用 makefirstkey 函数
    for (i = 0; i < readlen; i += 8)                    //循环
    {
        handle_data((ULONG32*) &data[i], DESDECRY);     //调用 handle_data 函数
    }
    return SUCCESS;                                     //返回 SUCCESS
}
```

（5）创建函数 des3()用于生成加密密钥，把待加密的明文分割成 64 位的块，把第 i–1 层加密后的密文作为第 i 层加密的明文输入，根据用户指定的加密层数进行 n 层加密，最终生成的密文存放在 data 所指向的内存中。第一个形式参数用于存放待加密明文的内存指针；第二个形式参数用于存放用户输入的密钥内存的指针；第三个形式参数为用户指定进行加密加密的层数；第四个形式参数为待加密明文的长度。函数代码如下。

```
INT32 des3(ULONG8 *data, ULONG8 *key, ULONG32 n, ULONG32 readlen)
{
    ULONG32 i = 0, j = 0;                               //声明整型变量，并设置初始值
    makefirstkey((ULONG32*)key); /*产生密钥*/           //调用 makefirstkey 函数
    for (i = 0; i < n; i++)                             //循环
    {
        for (j = 0; j < readlen; j += 8)                //循环
        {
            handle_data((ULONG32*) &data[j], DESENCRY); //调用 handle_data 函数
        }
    }
    return SUCCESS;                                     //返回 SUCCESS
}
```

（6）创建函数 Ddes3()用于生成解密密钥，把待解密文分割成 64 位的块，把第 i–1 层解密后的“明文”作为第 i 层解密的密文输入，根据用户指定的解密层数进行 n 层解密，最终生成的明文存放在 data 所指向的内存中。用户仅仅输入一条密钥，所有的解密密钥都是由这条密钥生成。第一个形式参数用于存放待解密文的内存指针；第二个用于存放用户输入的密钥内存的指针；第三个为用户指定进行解密的层数；第四个形式参数为待解密文的长度。函数代码如下。

```
INT32 Ddes3(ULONG8 *data, ULONG8 *key, ULONG32 n, ULONG32 readlen)
{
    ULONG32 i = 0, j = 0;                               //声明整型变量，并设置初始值
    makefirstkey((ULONG32*)key); /*产生密钥*/
```

```
    for (i = 0; i < n; i++)                                    //外层循环
    {
        for (j = 0; j < readlen; j += 8)                       //内层循环
        {
            handle_data((ULONG32*) &data[j], DESDECRY);        //调用 handle_data 函数
        }
    }
    return SUCCESS;                                            //返回 SUCCESS
}
```

（7）创建函数 desN()用于生成加密密钥，把待加密的明文分割成 64 位的块，把第 i–1 层加密后的密文作为第 i 层加密的明文输入，根据用户指定的加密层数进行 n 层加密，最终生成的密文存放在 data 所指向的内存中。用户通过输入的密钥条数决定加密的层数，每轮 16 次迭代加密所使用的加密密钥是由用户自定的对应密钥生成。第一个形式参数用于存放待加密明文的内存指针；第二个形式参数用于存放用户输入的密钥内存的指针；第三个形式参数为用户指定的密钥条数；第四个形式参数为待加密明文的长度。函数代码如下。

```
INT32 desN(ULONG8 *data, ULONG8 **key, ULONG32 n_key, ULONG32 readlen)
{
    ULONG32 i = 0;
    for (i = 0; i < n_key; i++)                                //循环
    {
        des(data, key[i], readlen);                            //调用 des 函数
    }
    return SUCCESS;                                            //返回 SUCCESS
}
```

（8）创建函数 DdesN()用于生成解密密钥，把待解密文分割成 64 位的块，把第 i–1 层解密后的”明文”作为第 i 层解密的密文输入，根据用户指定的解密层数进行 n 层解密，最终生成的明文存放在 data 所指向的内存中。用户通过输入的密钥条数决定解密的层数，每轮 16 次迭代加密所使用的解密密钥是由用户自定的对应密钥生成。第一个形式参数用于存放待解密文的内存指针；第二个形式参数用于存放用户输入的密钥内存的指针；第三个形式参数为用户指定密钥的条数；第四个形式参数为待解密文的长度。函数代码如下。

```
INT32 DdesN(ULONG8 *data, ULONG8 **key, ULONG32 n_key, ULONG32 readlen)
{
    INT32 i;                                                   //声明整型变量
    for (i = n_key; i > 0; i--)                                //循环
    {
        Ddes(data, key[i - 1], readlen);                       //调用 Ddes 函数
    }
    return SUCCESS;                                            //返回 SUCCESS
}
```

举一反三

根据本实例，读者可以：

- 使用 DES 算法的文本加密程序；

- 采用 DES 算法制作注册机程序。

算法 26 RC4 加密

实例说明

RC4 加密算法是大名鼎鼎的 RSA 三人组中的头号人物 Ron Rivest 在 1987 年设计的密钥长度可变的流加密算法簇。之所以称其为簇，是由于其核心部分的 S-box 长度可为任意，但一般为 256 字节。本实例用于介绍 RC4 加密的方法。运行本实例编译后的可执行文件，执行参数为“C:\11.txt”、“C:\12.txt”、“aj”，将文件 11.txt 进行 RC4 加密，生成文件 12.txt。程序运行结果如图 15.36 所示，加密前的文件内容如图 15.37 所示，加密后的文件内容如图 15.38 所示。

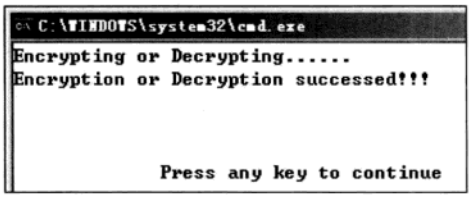


图 15.36 加密或解密过程

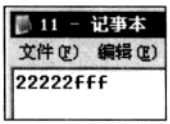


图 15.37 原文件

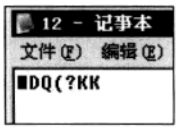


图 15.38 加密后

技术要点

RC4 算法包括初始化算法（KSA）和伪随机子密码生成算法（PRGA）两大部分。

1. 初始化

初始化 S-Box。首先，进行线性填充：S[0]=1,S[1]=1,...,S[255]=255。然后用密钥填充另一个 256 字节数组，不断重复密钥直至填充到整个数组中：K[0],K[1],...,K[255]。将指针 J 设为 0，然后：

I 从 0～255

$$J=(J+S[I]+K[I]) \bmod 256$$

交换 S[I]和 S[J]

2. 伪随机子密码生成

S-Box 的所有项都是数字 0～255 的置换，并且这个置换是一个可变长度密钥的函数。它有两个计数器：I 和 J，初始值为 0。

生成伪随机子密码步骤如下。

$$I=(I+1) \bmod 256$$

$$J=(J+S[I]) \bmod 256$$

交换 S[I]和 S[J]

$$T=(S[I]+S[J]) \bmod 256$$

$$K=S[T]$$

字节 K 为伪随即子密码，它与明文异或产生密文，或者与密文异或产生明文。

实现过程

例 15.26　RC4 加密（光盘位置：光盘\MR\Instance\15\26）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h、stdlib.h、string.h。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

（3）创建函数 re_RC4()，用于进行 RC4 算法的初始化与伪随机子密码的生成。过程 re_RC4 的一个参数用于指定数簇，第二个参数用于指定密钥。在初始化的过程中，密钥的主要功能是将一个 256 字节的初始数簇进行随机搅乱，不同的数簇在经过伪随机子密码生成算法的处理后可以得到不同的密钥序列。代码如下。

```
void re_RC4(unsigned char *S, char *key)                    //始化 RC4 算法
{
    char T[255] = {0};                                      //声明字符类型数组
    re_S(S);
    re_T(T, key);
    re_Sbox(S, T);
}

void re_S(unsigned char *S)
{
    unsigned int i;
    for (i = 0; i < 256; i++)
    //S[0] = 0, S[1] = 1...S[255] = 255
        S[i] = i;
}

void re_T(unsigned char *T, char *key)                      //把 key 循环放进 T
{
    int i;
    int keylen;
    keylen = strlen(key);
```

```
    for (i = 0; i < 256; i++)
        T[i] = key[i % keylen];
}

void re_Sbox(unsigned char *S, unsigned char *T)          //生成 S-Box
{
    int i;
    int j = 0;
    for (i = 0; i < 256; i++)
    {
        j = (j + S[i] + T[i]) % 256;
        swap(&S[i], &S[j]);
    }
}
```

（4）创建函数 RC4()，用于对指定的文件内容进行加密或解密，并生成新的文件。参数 readfile 用于指定源文件路径，参数 writefile 用于指定目标文件路径，参数 key 用于指定密钥。代码如下。

```
void RC4(FILE *readfile, FILE *writefile, char *key)
{
    unsigned char S[255] = {0};
    unsigned char readbuf[1];
    int i, j, t;
    re_RC4(S, key);                                       //初始化 RC4
    i = j = 0;
    while (fread(readbuf, 1, 1, readfile))
    {
        i = (i + 1) % 256;
        j = (j + S[i]) % 256;
        swap(&S[i], &S[j]);
        t = (S[i] + (S[j] % 256)) % 256;
        readbuf[0] = readbuf[0] ^ S[t];
        fwrite(readbuf, 1, 1, writefile);
        memset(readbuf, 0, 1);
    }
    printf("Encryption or Decryption successed!!!\n");
}
```

（5）在主函数 main()中，对程序的执行参数进行介绍，并调用函数 RC4()，进行文件的加密或解密操作。代码如下。

```
int main(int argc, char *argv[])
{
    char *Filename1,  *Filename2;
    char *key;
    FILE *file1,  *file2;
    Filename1 = argv[1];
    Filename2 = argv[2];
    key = argv[3];
    if (argc == 4)
    {
        printf("Encrypting......\n");
    }
    else
    {
        printf("parameters£° \n");
        printf("source | destination |key \n");
        return  - 1;
```

```
    }
    if ((file1 = fopen(Filename1, "rb")) == NULL || (file2 = fopen(Filename2,
        "wb")) == NULL)
    {
        printf("Can't open file\n");
        return  - 1;
    }
    RC4(file1, file2, key);
    fclose(file1);
    fclose(file2);
    return 0;
}
```

举一反三

根据本实例，读者可以：

- 采用 RC4 算法制作软件授权文件；
- 采用 RC4 算法加密用户登录密码。

算法 27　打渔晒网问题

实例说明

如果一个渔夫从 2000 年 1 月 1 日开始每三天打一次鱼，两天晒一次网，编程实现当输入 2000 年 1 月 1 日以后的任意一天，输出该渔夫是在打渔还是在晒网？运行结果如图 15.39 所示。

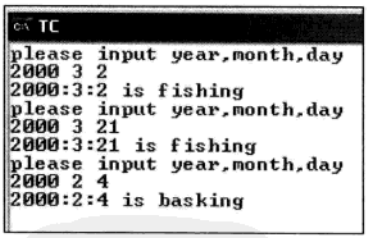


图 15.39　打渔晒网问题

技术要点

要实现本实例要求的功能主要有以下两个技术要点。

第一，判断输入的年份（2000 年以后包括 2000 年）是否是闰年，这里我们自定义函数 leap()来进行判断。该函数的核心内容就是闰年的判断条件即能被 4 整除但不能被 100 整除，或能被 400 整除。

第二，如何求输入日期距 2000 年 1 月 1 日有多少天。首先是判断 2000 年距输入的年份间有多少年，这其中有多少年是闰年就加多少个 366，有多少年是平年也同样加多少个 365，其次是要将 12 个月每月的天数存到数组中，因为闰年 2 月份的天数有别于平年，故采用两个数组 a 和 b 分别存储。当输入年份是平年，月份为 m 的时侯就在前面累加的日期的基础上继续累加存储着平年每月天数的数组的前 $m-1$ 个元素，将累加的结果加上输入的日期便求出了最终结果；闰年的算法类似。

实现过程

例 15.27 打渔晒网问题（光盘位置：光盘\MR\Instance\15\27）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）自定义 leap()函数，用来判断输入的年份是否是闰年，代码如下。

```
int leap(int a)                                          /*自定义函数 leap 用来指定年份是否为闰年*/
{
    if (a % 4 == 0 && a % 100 != 0 || a % 400 == 0)      /*闰年判定条件*/
        return 1;                                        /*不是闰年返回 1*/
    else
        return 0;                                        /*不是闰年返回 0*/
}
```

（4）自定义 number()函数，用来计算输入日期距 2000 年 1 月 1 日共有多少天，代码如下。

```
int number(int year, int m, int d)          /*自定义函数 number 计算输入日期距 2000 年 1 月 1 日共有多少天*/
{
    int sum = 0, i, j, k, a[12] =
    {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };                                      /*数组 a 存放平年每月的天数*/
    int b[12] =
    {
        31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };                                      /*数组 b 存放闰年每月的天数*/
    if (leap(year) == 1)                    /*判断是否为闰年*/
        for (i = 0; i < m - 1; i++)
            sum += b[i];                    /*是闰年，累加数组 b 前 m-1 个月份天数*/
    else
        for (i = 0; i < m - 1; i++)
            sum += a[i];                    /*不是闰年，累加数组 a 钱 m-1 个月份天数*/
    for (j = 2000; j < year; j++)
        if (leap(j) == 1)
            sum += 366;                     /*2000 年到输入的年份是闰年的加 366*/
    else
        sum += 365;                         /*2000 年到输入的年份不是闰年的加 365*/
    sum += d;                               /*将前面累加的结果加上日期，求出总天数*/
    return sum;                             /*将计算的天数返回*/
}
```

（5）main()函数作为程序的入口函数，代码如下。

```
main()
{
    int year, month, day, n;
    printf("please input year,month,day\n");
    scanf("%d%d%d", &year, &month, &day);          /*输入年月日*/
    n = number(year, month, day);                  /*调用函数 number*/
    if ((n % 5) < 4 && (n % 5) > 0)                /*当余数是 1 或 2 或 3 时说明在打渔否则在晒网*/
```

```
            printf("%d:%d:%d is fishing\n", year, month, day);
        else
            printf("%d:%d:%d is basking\n", year, month, day);
    }
```

举一反三

根据本实例，读者可以：

- 如果一个渔夫从 1990 年 1 月 1 日开始每三天打一次鱼两天晒一次网，编程实现当输入 1990 年 1 月 1 日以后的任意一天，输出该渔夫是在打渔还是在晒网；
- 编程实现输入 1990 年后的某一年，输出该年的日历。

算法 28　水池注水问题

实例说明

有 4 个水渠（A、B、C、D）向一个水池注水，如果单开 A，3 天可以注满；如果单开 B，1 天可以注满；如果单开 C，4 天可以注满；如果单开 D，5 天可以注满。问如果 A、B、C、D 这 4 个水池同时注水，注满水池需要几天？运行结果如图 15.40 所示。

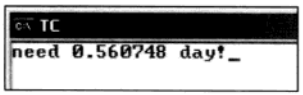


图 15.40　水池注水问题

技术要点

首先要求出一天每渠的注水量，这里分别是 1/3、1/1、1/4、1/5，要 4 渠共同注水就求出每天注水之和即 1/3+1/1+1/4+1/5，那么 1 池水多久能注满只需用 1 除以它们的和即可。

实现过程

例 15.28　水池注水问题（光盘位置：光盘\MR\Instance\15\28）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件 stdio.h。

```
#include<stdio.h>
```

（3）定义变量类型为单精度型，计算出 4 渠每天注水量之和，再用 1 除以 4 渠每天的注水量，就可求出多久注满一池水。

（4）主要程序代码如下。

```
main()
{
    float a1 = 3, b1 = 1, c1 = 4, d1 = 5;                    /*定义变量为单精度型*/
    float day;                                               /*定义天数为单精度型*/
    day = 1 / (1 / a1 + 1 / b1 + 1 / c1 + 1 / d1);

                                                             /*计算 4 渠同时注水多久可以注满*/
    printf("need %f day!", day);                             /*将计算出的天数输出*/
}
```

举一反三

根据本实例，读者可以：

- 在本实例基础上编程计算如果只开 A 和 B 两个水渠进行注水，多久可以把水池注满；
- 在本实例基础上编程计算如果只开 C 和 D 两个水渠进行注水，多久可以把水池注满。

算法 29 迪杰斯特拉算法

实例说明

用迪杰斯特拉算法求解图 15.41 顶点 1 到各顶点的最短路径，运行结果如图 15.42 所示。

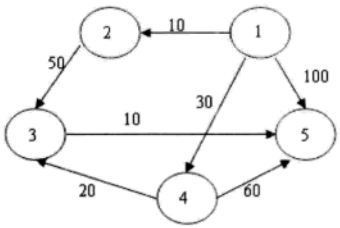


图 15.41　求最短路径

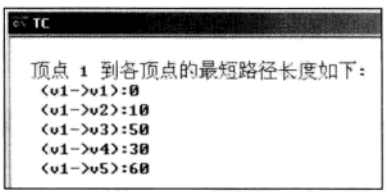


图 15.42　迪杰斯特拉算法

技术要点

迪杰斯特拉算法的思想是：设置并逐步扩充一个集合 S，存放已求出其最短路径的顶点，则尚未确定最短路径的顶点集合是 V-S，其中 V 为网中所有顶点集合。按最短路径长度递增的顺序逐个将 V-S 中的顶点加到 S 中，直到 S 中包含全部顶点，而 V-S 为空。

实现过程

例 15.29　迪杰斯特拉算法（光盘位置：光盘\MR\Instance\15\29）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件并进行宏定义及全局变量声明。

```
#include <stdlib.h>
#include <stdio.h>
#define MAXNODE 30
#define MAXCOST 1000
int dist[MAXNODE], cost[MAXNODE][MAXNODE], n = 5
```

（3）自定义 prim()函数，用来求最小生成数，代码如下。

```
void dijkstra(int v0)
{
    int s[MAXNODE];
    int mindis, dis, i, j, u;
    for (i = 1; i <= n; i++)
    {
        dist[i] = cost[v0][i];
        s[i] = 0;
    }
    s[v0] = 1;
    for (i = 1; i <= n; i++)
    {
        mindis = MAXCOST;
        for (j = 1; j <= n; j++)
        if (s[j] == 0 && dist[j] < mindis)
        {
            u = j;
            mindis = dist[j];
        }
        s[u] = 1;
        for (j = 1; j <= n; j++)
        if (s[j] == 0)
        {
            dis = dist[u] + cost[u][j];
            dist[j] = (dist[j] < dis) ? dist[j]: dis;
        }
    }
}
```

（4）自定义 display_path()函数，用来输出到各顶点的最短路径，代码如下。

```
void display_path(int v0)
{
    int i;
    printf("\n 顶点 %d 到各顶点的最短路径长度如下:\n", v0);
    for (i = 1; i <= n; i++)
    {
        printf("  (v%d->v%d):", v0, i);
        if (dist[i] == MAXCOST)
            printf("wu lun jing\n");
        else
            printf("%d\n", dist[i]);
    }
}
```

（5）主要程序代码如下。

```
main()
{
    int i, j, v0 = 1;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            cost[i][j] = MAXCOST;
    for (i = 1; i <= n; i++)
        cost[i][i] = 0;
    cost[1][2] = 10;
    cost[1][5] = 100;
    cost[1][4] = 30;
    cost[2][3] = 50;
    cost[4][3] = 20;
    cost[3][5] = 10;
    cost[4][5] = 60;
    dijkstra(v0);
    display_path(v0);
}
```

举一反三

根据本实例，读者可以：

- 用迪杰斯特拉算法求解图 15.43 中顶点 1 到各顶点的最短路径。

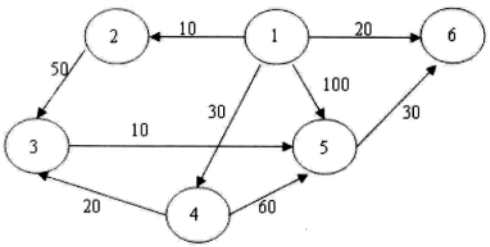


图 15.43　求最短路径

算法 30　百钱买百鸡

实例说明

中国古代数学家张丘建在他的《算经》中提出了一个著名的“百钱买百鸡问题”，鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一；百钱买百鸡，问翁、母、雏各几何？运行结果如图 15.44 所示。

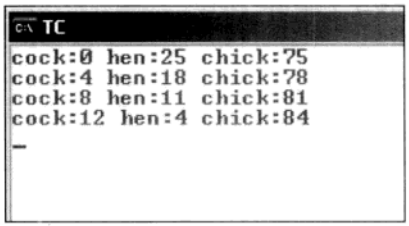


图 15.44　百钱买百鸡问题

技术要点

根据题意设公鸡、母鸡和雏鸡分别为 cock、hen 和 chick，如果 100 元全买公鸡那么最多能买 20 只，所以 cock 的范围是大于等于 0，小于等于 20，如果全买母鸡那么最多能买 33 只，所以 hen 的范围是大于等于 0，小于等于 33，如果 100 元钱全买小鸡那么根据题意最多能买 99 只（根据题意小鸡的数量应小于 100 且是 3 的倍数）。在确定了各种鸡的范围后进行穷举并判断，判断的条件有以下 3 点。

- 所买的 3 种鸡的钱数总和为 100。
- 所买的 3 种鸡的数量之和为 100。
- 所买的小鸡数必须是 3 的倍数。

实现过程

例 15.30　百钱买百鸡（光盘位置：光盘\MR\Instance\15\30）。

（1）在 TC 中创建一个 C 文件。

（2）引用头文件。

```
#include <stdio.h>
```

（3）使用 for 语句对 3 种鸡的数量在事先确定好的范围内进行穷举并判断，对满足条件的将 3 种鸡的数量按指定格式输出，否则进行下次循环。

（4）主要程序代码如下。

```
main()
{
    int cock, hen, chick;                                    /*定义变量为基本整型*/
    for (cock = 0; cock <= 20; cock++)                       /*公鸡范围在 0 到 20 之间*/
        for (hen = 0; hen <= 33; hen++)                      /*母鸡范围在 0 到 33 之间*/
            for (chick = 3; chick <= 99; chick++)            /*小鸡范围在 3 到 99 之间*/
                if (5 *cock + 3 * hen + chick / 3 == 100)    /*判断钱数是否等于 100*/
```

```
                if (cock + hen + chick == 100)              /*判断购买的鸡数是否等于100*/
                    if (chick % 3 == 0)                     /*判断小鸡数是否能被3整除*/
                        printf("cock:%d hen:%d chick:%d\n", cock, hen,chick);
}
```

举一反三

根据本实例，读者可以：

- 围绕一个山顶有 10 个洞，一只兔子和一只狐狸各居一洞，狐狸第一次隔一个洞找兔子，第二次隔两个洞找兔子，依此类推，假使狐狸找 1 000 次，问那个洞是安全的；
- 选择不同日期，限制不同时间，实现定时开关机。